968

A155599

A04 155 599

R&D CENTER

LABORATORY

TECHNICAL REPORT

NO. 13073

EVALUATION OF A SOFTWARE DESIGN APPROACH (DEVELOP A THREAT RESOLUTION MODULE IN Ada)

CONTRACT NO. DAAE07-83-C-R056

5 MARCH 1985

by S.J. WERSAN, M.T.MINOGUE

DALMO VICTOR OPERATIONS
Bell Aerospace TEXTRON

1515 Industrial Way
Belmont, California 94002
Tel. (415) 595-1414

U.S. ARMY TANK-AUTOMOTIVE COMMAND
RESEARCH AND DEVELOPMENT CENTER
Warren, Michigan 48090

R-3811-10968A

NOTICES

The findings in this report are not to be construed as an official Department of the Army position.

Mention of any trade names or manufacturers in this report shall not be construed as advertising nor as an official endorsement or approval of such products or companies by the U.S. Government.

This report consists mainly of information researched from material in the public domain plus developmental techniques originated by Dalmo Victor under the subject contract. However, the Operating System software used on the workstations and the Ada code compiler must remain proprietary to the commercial developer and vendor, Callan Data Systems (AT&T UNIX) and TeleSoft Corporation (Ada Compiler for the MC68000 under UNIX), respectively.

Destroy this report when it is no longer needed. Do not return to the originator.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY DD-254 dated 8 November 1984	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE --	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) R-3811-10968A	5. MONITORING ORGANIZATION REPORT NUMBER(S) 13073
6a. NAME OF PERFORMING ORGANIZATION Dalmo Victor-Textron 6b. OFFICE SYMBOL (*If applicable*) --	7a. NAME OF MONITORING ORGANIZATION U.S. ARMY TACOM R & D Center
6c. ADDRESS (*City, State, and ZIP Code*) 1515 Industrial Way Belmont, California 94002	7b. ADDRESS (*City, State, and ZIP Code*) Warren, Michigan 48090
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*) AMSTA-ZSC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAE07-83-C-R056
8c. ADDRESS (*City, State, and ZIP Code*) Warren, Michingan 48090	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Evaluation of a Software Design Approach: Develop a Threat Resolution Module in Ada[R].

12. PERSONAL AUTHOR(S) Wersan, Dr. Stephen J., Minogue, Marie T., Simmen, Robert L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Final	FROM 4-83 TO 11-84	1985, March 5	118

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	
17	02		Data Fusion, Run Time Software Applications, Ada[R] Higher Order Language, Threat Resolution, UNIX Operating System, Multispectral Sensor Integration

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The Vehicle Integrated Defense System (VID) incorporates several packages of application software written in Ada[R]. This report describes the preliminary design and code generation of one of the critical packages, the Threat Resolution Module (TRM). The necessary logic design, table, and file creation and other tasks to implement a preliminary TRM was accomplished. All programming was written in the DOD standard Higher Order Language of Ada[R]. Work was done on the UNISTAR 200 workstation hosting an Ada[R] Compiler developed by TeleSoft Corporation and targeting the MC68000 microcomputer. The operating system for the software development was UNIX. This report describes the feasibility of the VIDS-DMS software design approach by discussing the three principal topics of:

1. Software algorithm design technology
2. Code generation, edit, and debug using Ada[R] as the PDL
3. Test of the preliminary TRM on a skeleton operating system (testbed)

(continued on Page 2)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Francis B. Hoogterp	22b. TELEPHONE (*Include Area Code*) (313)574-6693 22c. OFFICE SYMBOL AMSTA-ZSC

The annotated source code of the software as well as the object code for the TRM is delivered under separate cover with this report to TACOM. This data, in conjunction with the "lessons learned" during the experimental development project form the baseline for continued development of the VIDS-DMS Feasibility Demonstration Model software.

SUMMARY. Work was carried out which demonstrated the feasibility of Ada Higher Order Language (HOL) as a Program Design Language (PDL) for the Vehicle Integrated Defense System - Data Management System (VIDS-DMS).

A preliminary, non-real time version of a specific application package of the VIDS-DMS, the Threat Resolution Module, was designed, written, coded, debugged, and tested. This demonstrated the ability of Ada HOL and the Ada PDL environment to support the development of a multisensor integration program.

The Threat Resolution Module (TRM) is the critical application package of the VIDS-DMS in that it is responsible for detection, tracking, correlation, and reaction decision functions. All other application modules to be developed in Phase III are subordinate to the TRM.

Once the complex nature of the language was mastered, it was an efficient method of organizing file structures and designing the overall program. Structured English provided great visibility and ease of demonstration. Multiuser workstations operating under the UNIX operating system proved effective even though the Ada compiler was not a complete implementation. Workarounds resulted in operable code which could be tested and demonstrated on a standalone testbed developed on this project for this purpose.

Lessons learned and recommendations for further development are included in the report with examples of the structured English source documents which yielded the Ada operational code targeted to the MC68000 microprocessor. These lessons will be applied toward the creation of a real-time version of the Threat Resolution Module and the rest of the VIDS software suite in the succeeding stages of the VIDS Program.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0. INTRODUCTION

This report describes the results of an experimental software design and develpment program for the TACOM Vehicle Integrated Defense System by Dalmo Victor-Textron under Contract No. DAAE07-83-C-R056. The heart of the VIDS is its Data Management System (DMS) which processes information from the threat detection sensors and then initiates the best reaction to counter the threat. The key element of the DMS software is the Threat Resolution Module which accepts the raw sensor output and detects, classifies, and prioritizes the threats. The work presented here will contribute to the enhancement of modern armored vehicle survivability and thus increased effectiveness, particularly in an environment of a numerically superior opposition.

The need for this segment of the program was brought about by the (then) lack of experience by anyone in industry or the government with the requirements for:

- Multispectral sensor integration, and
- Real time processing of application software written in the Ada Higher Order Language.

Thus, the project set about to prove the feasibility of not only developing algorithms for the necessary tracking and correlation of threat sensors, but also the efficacy of coding the algorithms using the new DOD standard Ada as the programming design language.

Such algorithms have since been developed, written in Ada, coded, and tested. The results accompany this technical report. Program development was carried out on a Callan Data Systems Workstation in which the host processor is an MC68000 CPU. Testing was demonstrated in which the object code is targeted on the MC68000 CPU. This Threat Resolution Module is a preliminary of the eventual TRM algorithm(s) to be developed for the VIDS Data Management System (VIDS-DMS) and has shown the difficulties and the promises of Ada and the MC68000 as realtime multitasking software and hardware.

2.0. OBJECTIVES

The overall objective of this project was to experimentally evaluate a key software module. To meet this objective, the contractor did the necessary logic design, table, and file creation and other tasks to implement a preliminary VIDS-DMS Threat Resolution Module (TRM) to meet the general intent of Section 3.3.2.8 of the VIDS-DMS FDM Specification as described in Dalmo Victor Report No. R-3710-10307. To accomplish this task, the following specific objectives were met:

- o Preliminary TRM designed.
- o The TRM code was tested to demonstrate its operation.
- o Programming was carried out in Ada Higher Order Language using the UNISTAR/UNIX operating system software development workstation, hosting an Ada compiler supplied by TeleSoft Corporation.
- o Testing of the TRM was carried out with the Dalmo Victor Threat Engagement Scenario Simulator Model dedveloped in 1982 on IR & D funds.
- o Documentation was carried out and delivered per the Statement of Work.

This report provides the following information on the project:

- o Description of the technical effort
- o Discussion of lessons learned during the development
- o Annotated descriptions of the source code
- o Results of TRM development on a 5-1/4-inch floppy disk.

Since the TRM is a subset of the overall DMS software and must operate in the absence of the eventual sensors and the DMS Operating System Executive, we also describe the deviations expected between this TRM and the eventual TRM used in the FDM. Additionally, since the TRM must be tested in the absence of the actual DMS operating system, a special TRM testbed was developed for test of the preliminary TRM. This testbed was based on the UNIX operating system and was programmed in Ada using the partial Ada compiler developed by TeleSoft. Commentary on this incomplete compiler and the expectations for Phase III work using all facilities of a fully-validated compiler from TeleSoft are also provided in the Recommendations section of the report.

An illustration of the overall software development operation and the Ada programming workstation on which the TRM and the Engagement Model are hosted are shown in Figure 2-1.

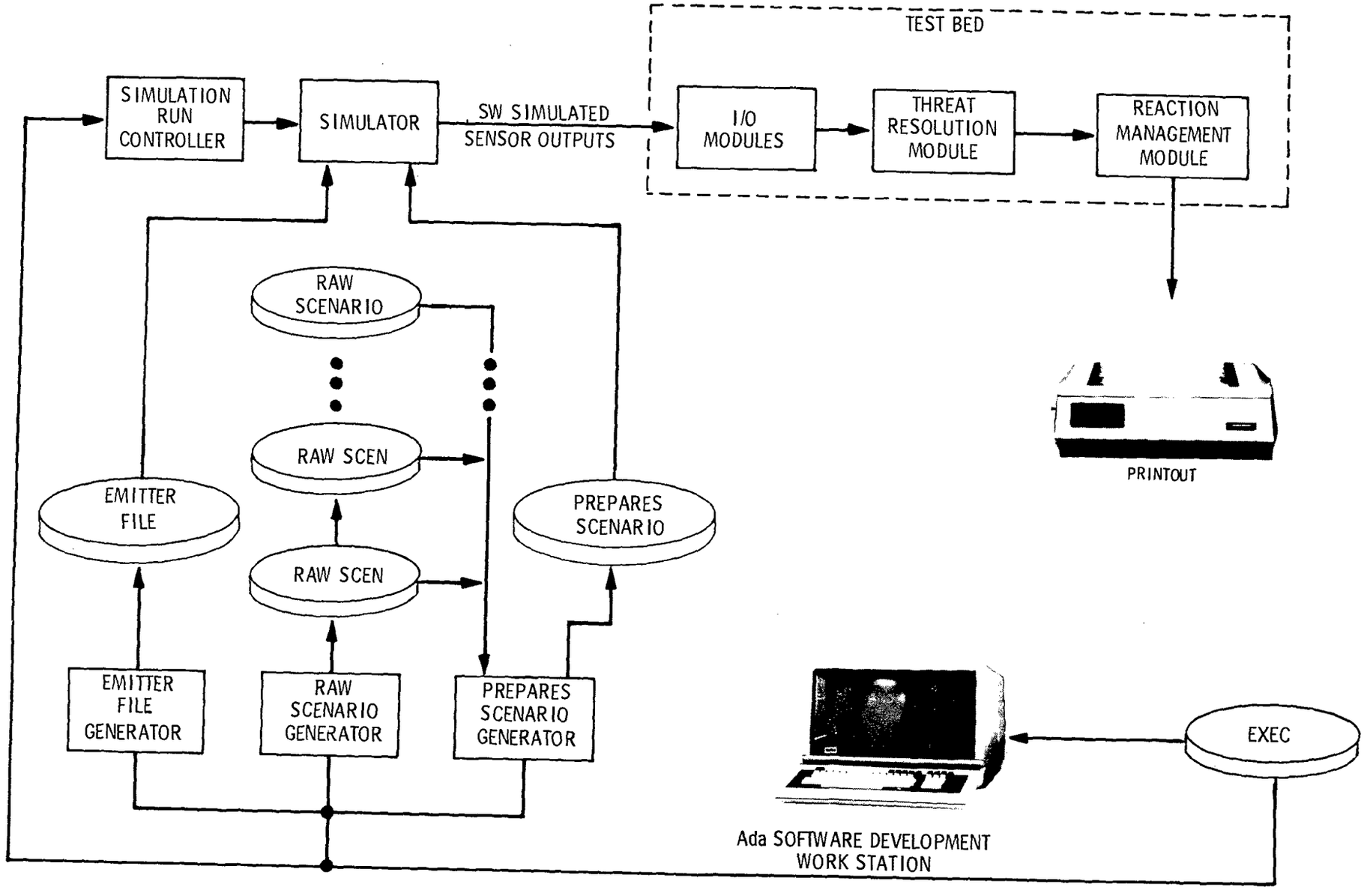


Figure 2-1. TRM Test Configuration

3.0. CONCLUSIONS

Ada HOL is a feasible Programming Design Language. Once the complexities of the language are mastered, it is simple to write and debug code. It is appears to be a good implementation language once a satisfactory development environment is established.

Ada programs are easier to code and debug than other higher order languages (FORTRAN, COBOL, PL/M) and should facili-tate development of larger programs. They should also live up to expectations of ability to redesign and recode in the field. Although there is no proof of this, there is evidence based on the experience of several people working on parts of the same module showing reliable interchange-ability of sequential versions of code.

We have successfully developed a preliminary model of the Threat Resolution Module which serves as a baseline for the nucleus of the software application package for the FDM VIDS-DMS.

The software program and support structure is a workable model. It is flexible, trainable (if trainee has sufficient structured programming experience) and can be adapted to other projects through a revision of the application modules as appropriate to the functional requirements of the new system.

A more complete set of software development tools is required. Especially useful would be a tool to keep track of dependencies of modules. This tool must distinguish between the specification and body of a package.

Future Ada programmers should be well schooled in the full language not only syntax.

There were problems in the development of the present software, but these were not due to the Ada language or the VIDS-DMS system for the most part. One source of difficulty beyond our control was the incomplete nature of the TeleSoft Ada compiler, a pre-validation release. Particularly missed were the Ada features dealing with "generics" and the complete set of features concerned with "tasking."

Several workaround techniques were necessary and inefficiencies resulted in significant stretch-out of the project schedule. These inefficiencies will be eliminated when the completely validated Ada compiler is employed on future programming activities.

Our use of the language at this point has proven that we must carefully evaluate Ada features and abilities when making design decisions.

4.0 RECOMMENDATIONS

A. Continue development of VIDS-DMS software using the modified family tree structure in Figure 5-2.

B. Get a mature compiler that permits compiling a package body separately from its specification (separate compilation feature).

C. Study features of Ada "tasking" and attempt to transfer TRM algorithms into tasks during Phase III development.

D. Eliminate KLUDGE software by using full Ada compiler implementation.

E. Use Ada "generic" routines in Phase III to take advantage of strong type characteristics of Ada. Reduce volume of source code by declaring program templates.

F. Enforce rule of avoiding the "Use" statement thereby reducing confusion to programmer during debug and program maintenance.

G. Place an exception handler in all subprograms and packages and add block to all I/O requests so that exception handler can be included.

H. Develop a more complete set of software development tools. Especially useful would be a tool to keep track of dependencies of modules. This tool must be able to distinguish between the specification and body of a package.

I. School future Ada programmers well in the full language; not only syntax.

J. Use "No Break" (UPS) power supplies when using small standalone software development workstations with high capacity Winchester hard disks.

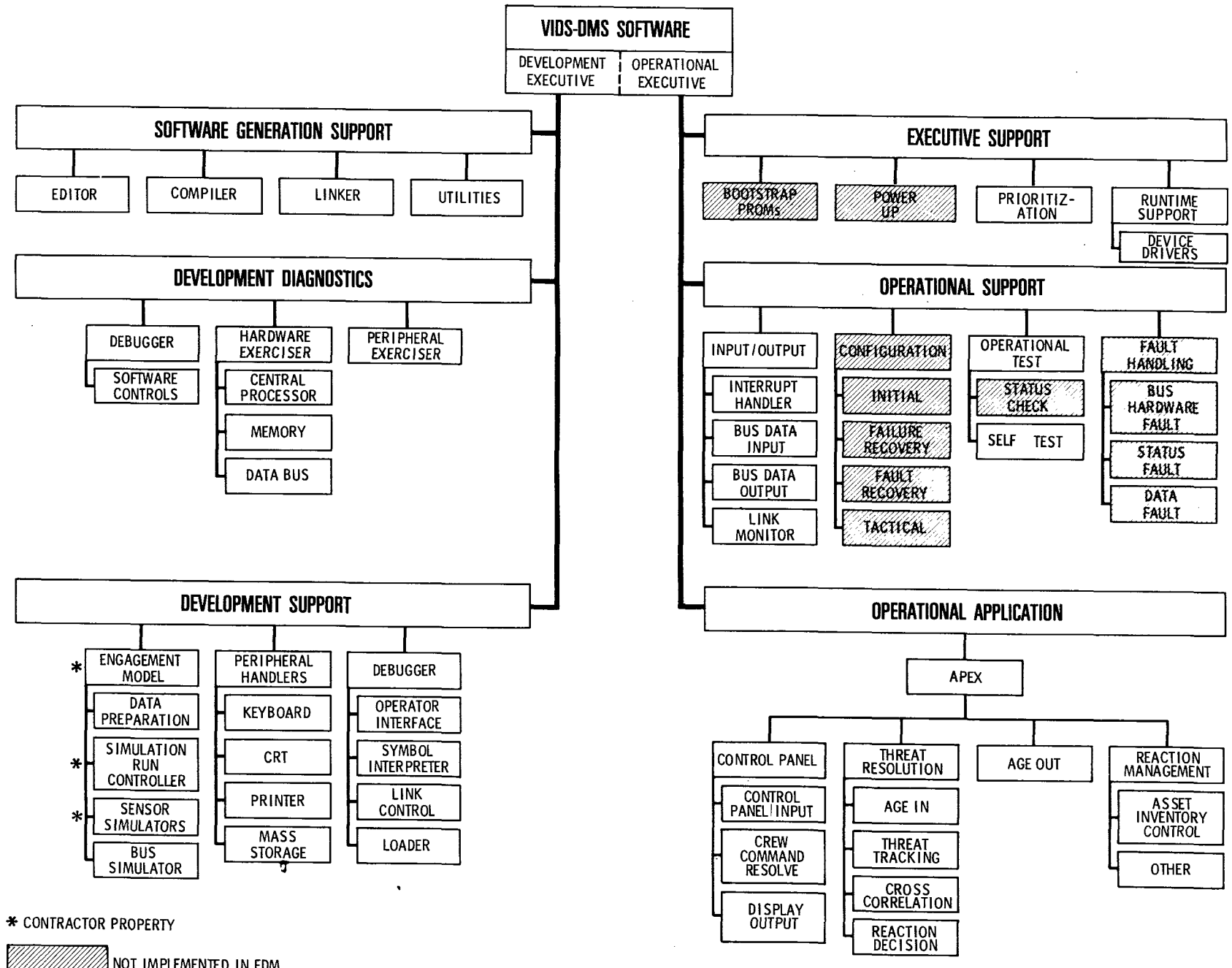


Figure 4-1. VIDS-DMS FDM Software Family Tree

5.0. DISCUSSION

5.1. Overview

Three issues of technical consideration were addressed:

- o Software design methodology
- o Code generation, edit and debug using Ada
- o Test of the TRM on a skeleton operating system or testbed

These issues form the basis of the investigation of this project and are discussed in this section.

An overview of the Risk Reduction software development for VIDS-DMS is illustrated in Figure 5-1. (The overall VIDS-DMS FDM software family tree is shown for reference only in Figure 4-1.) Description of the principal processes of the TRM are presented in Section 5-1 as are descriptions of the nature of static and dynamic file generation. Section 5-2 describes the actual practice and lessons learned in Ada code generation while Section 5-3 describes the testbed and the TRM.

The detailed source code listings resulting from this development are attached as an Appendix to this report. The actual source and object code is delivered on magnetic media (5-1/4-inch floppy disk) under separate cover as CLIN 0003.

5.1.1. Testbed Software System Overview. The testbed provides the operating environment for the TRM necessary to support the proper execution of the TRM. It provides the mechanism to supply data to the TRM and accepts the output from the TRM. Other support packages of the testbed provide data definitions and a time simulator. This section gives an overall description of the testbed and its interface with the TRM. A more detailed description of the testbed module is supplied in Section 5-4.

The testbed was designed to consist of three major categories of packages - one category for library functions and declarations, one for utility functions used by individual processes, and one for processes and subprocesses. The structure of the packages is shown in Figure 5-2.

5.1.1.1. Library Packages.

a. Mathematical Library. The mathematical library included the mathematical functions for statistics and trigometry. These functions are general purpose and could be released by any Ada software. We successfully reported these routines from the environment model TESS to the VIDS testbed development. The mathematical library consisted of the following modules: MATH_TYPES, ELEM_FUNC, STAT_FUNC, and TRIG_FUNC. Each module is described in the following text:

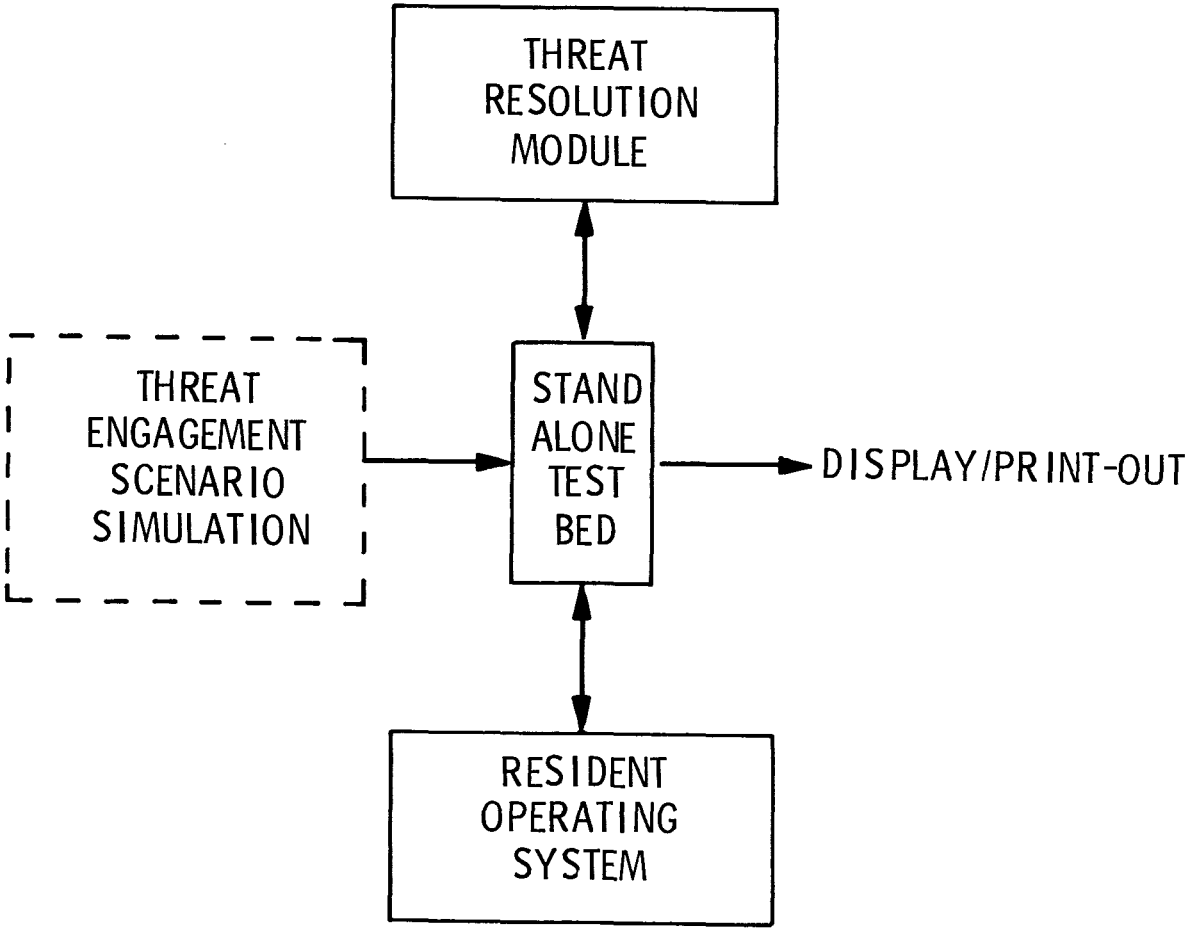


Figure 5-1. VIDS-DMS Risk Reduction Software Development

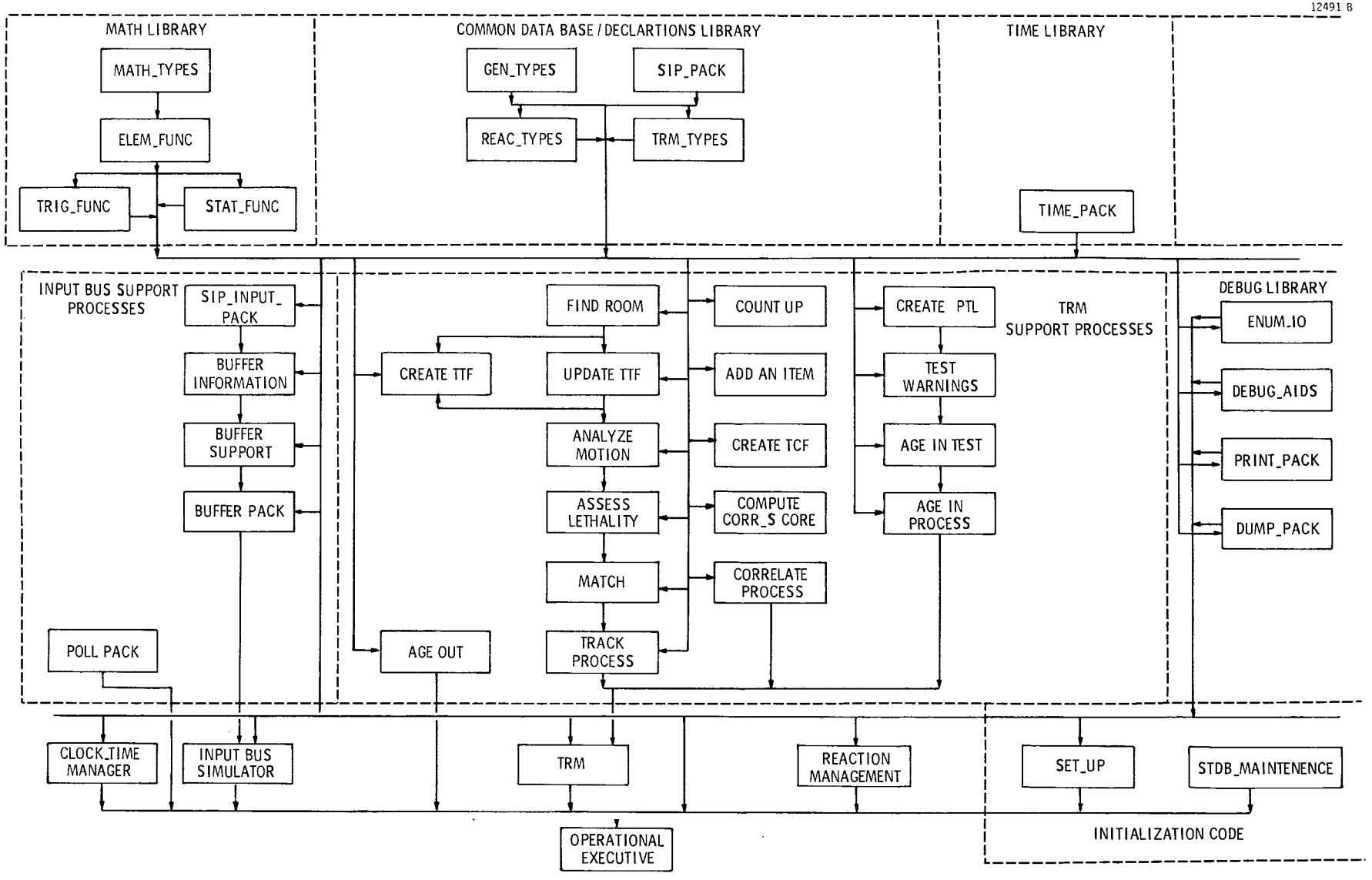


Figure 5-2. Test Bed and TRM Modules

- o MATH_TYPES - Defines commonly used type declarations which support the mathematical library functions found in the other modules. This module is accessed by all other mathematical modules.

- o ELEM_FUNC - Declares elementary mathematical functions such as square root, log, minimum and maximum. These functions all operate on floating point numbers and return floating point numbers.

- o STAT_FUNC - Defines probability functions for the normal distribution and a random number generator. This routine is used by the environment model TESS. The module is not a part of the testbed and need not be compiled with the testbed.

- o TRIG_FUNC - Defines the trigometric function for sine, cosine, and arctangent. It also defines the constants for pi/2, degrees per radian, and radians per degree.

b. Common Data Type Library. This library consists of the declarations of the commonly used types in the testbed. It is broken up into three subpackages: GEN_TYPES, SIP_PACK, and REAC_ TYPES. These packages are described in this section.

NOTE: Due to a compiler restriction, we are forced to group GEN_TYPES and SIP_PACK into one package called TYPE_LIBRARY.

- o GEN_TYPES - Defines miscellaneous data types for the VIDS testbed. This includes definitions for the SENSOR_ TYPES, EMITTER_TYPES and the numerical types for degrees and meters.

- o SIP_PACK - Defines the sensor input packet (SIP) for the input data. It also includes the definition of the INPUT_DATA_BLOCK used to transfer the input data from one testbed module to another.

- o REAC_TYPES - Defines the data types concerned with reaction decision and management. This also includes the definition of the THREAT_INFORMATION_BLOCK used to transfer the results of the TRM processing to other testbed modules.

c. Time Library. This package contains the definitions of the type for time and the legal functions that can be done on objects of type time. These functions include addition, subtraction, and comparison of two time varibles. It also includes multiplication of a floating point number to a time variable. The following utility functions are defined:

- o GET_TIME and PUT_TIME - Provide input/output routines for time variables.

- o CONVERT_TO_TIME - Converts a floating point or integer number to a time value.

- NEXT_SECOND - Returns a time equal to the next whole second based on the time argument passed to the function.

- WHOLE_SECONDS - Returns a time equal to the whole second portion of the input time.

- FRACTIONAL_SECONDS - Returns a time equal to fractional portion of the input time.

5.1.1.2. Utility Packages. The utility packages are used to group subprocesses into smaller, more manageable pieces as well as to create a database management module which declares the data type and its operations. This aided in the integration and debug because it reduced compile time. The TRM and the INPUT_BUS_SIMULATOR both used utility packages. The other processing modules did not require them. The utility packages are described in the discussions of the TRM and INPUT_ BUS_SIMULATOR. These packages are listed in Table 5-1.

5.1.1.3. Processes and Subprocesses Package. The process and subprocesses constitute the major subsections of the testbed and TRM. These processes are illustrated in Figure 5-2. The main processing unit is the OPERATIONAL_EXECUTIVE. It is responsible for all scheduling of all other processes and initializing the system. The remainder of processes shown in the figure are subprocesses to operational executive. The TRM is a drop-in module called by the OPERATIONAL_EXECUTIVE which supplies the data and all parameters required. The INPUT_BUS_SIMULATOR reads the SIP_FILES and passes SIPs to the OPERATIONAL_EXECUTIVE at the appropriate polling times. The CLOCK_TIME_MANAGER provides a pseudo clock and increments time. The REACTION_MANAGEMENT module provides the mechanism for reporting the results from the TRM. Each of these modules is described more fully in Section 5-1 (TRM) and Section 5-3 (Testbed).

5.2. VIDS Phase II Software Documentation

5.2.1. Introduction and Overview of Dynamic Data File. The purpose of this section is to introduce the structures, contents, terminology and routines connected with the internal, dynamic data files of the VIDS Phase II software. The design of the Phase II Threat Resolution Module (TRM) is centered about these data structures and their contents. So, this section of the documentation should be read before delving into the lower-level details of the TRM's constituent routines. The file handling routines to be discussed below are, of course, a part of these constituent routines, but deal with a level of detail that has little to do with the TRM's functional purposes, and are best understood via graphic rather than verbal illustration.

There are three principal types of internal, dynamic data files created and maintained by the TRM. These are the Threat Tracking Files (TTF), the Threat Correlation Files (TCF) and the Prioritized Threat List (PTL). The basic data types used to define and refer to these file types and to their components are exported from package TRM_TYPES (file name: trm types.text).

Table 5-1. Utility Packages

PACKAGE NAME	PARAGRAPH WHERE DISCUSSED	PAGE	MEMBER OF PACKAGE	SYSTEM FILE NAME
AGE_IN_PACK	5.2.3.5.	87		AGIN_PACK.TEXT
AGO_PACK	5.2.3.7.	96		AGO_PACK.TEXT
BUFFER_INFO	5.4.5.3.	107	BUFFER	BUFF.TEXT
BUFFER_SUPPORT	5.4.5.4.	107	BUFFER	BUFF.TEXT
BUFFER_PACK	5.4.5.5.	107	BUFFER	BUFF.TEXT
CLOCK_TIME_MANAGER	5.4.7.	107		CLOCK.TEXT
CORRELFILE	5.2.1.2.	35	DATA_FYLZ	DATA_FYLZ.TEXT
CORR_PACK	5.2.3.4.	77		CORR_PACK.TEXT
CREC_HANDLER	5.2.1.5.	48	DATA_FYLZ	DATA_FYLZ.TEXT
DEBUG_AIDS	5.4.9.2.	108	DEBUG_TOOLS	DBUG_TOOL.TEXT
DUMP_PACK	5.4.9.4.	108	DEBUG_TOOLS	DBUG_TOOL.TEXT
ELEM_FUNC	5.4.2.2.	105		ELEM_FUNC.TEXT
EMITTER_SETS	5.2.3.8.	98	SETS_PACK	SETS_PACK.TEXT
ENUM_IO	5.4.9.1.	108	DEBUG_TOOLS	DBUG_TOOL.TEXT
GEN_TYPES	5.4.3.1.	105	TYPE_ LIBRARY	TYPE_LIB.TEXT
INPUT_BUS_SIMULATOR	5.4.6.	107		INPUT_BUS.TEXT
MATH_TYPES	5.4.2.1.	105		MATH_TYPES.TEXT
OPERATIONAL_EXECUTIVE	5.4.11.	109		OP_EX.TEXT
POLL_PACK	5.4.5.1.	107		POLL_PACK.TEXT
PRINT_PACK	5.4.9.3.	108	DEBUG_TOOLS	DBUG_TOOL.TEXT
PRIOTHLIST	5.2.1.3.	44	DATA_FYLZ	DATA_FYLZ.TEXT
REACTION_MANAGEMENT	5.4.8.	108		REAC_MAN.TEXT
REAC_PACK	5.1.1.1.	18	NEW_TRM	NEW_TRM.TEXT
REAC_TYPES	5.4.3.3.	106		REAC_TYPES.TEXT
SENSOR_SETS	5.2.3.8.	98	SETS_PACK	SETS_PACK.TEXT
SET_UP	5.4.10.1.	108	TUNE_UP	TUNE_UP.TEXT
SIP_INPUT_PACK	5.4.5.2	107	BUFFER	BUFFER.TEXT
SIP_PACK	5.4.3.2.	106	TYPE_ LIBRARY	TYPE_LIB.TEXT
STATIC_DATABASE	5.2.2.	50		ST_DATA.TEXT
STDB_MAINTENANCE	5.4.10.2.	109	TUNE_UP	TUNE_UP.TEXT
STAT_FUNC	5.4.2.4.	105		STAT_FUNC.TEXT
THREATFILE	5.2.1.	19	DATA_FYLZ	DATA_FYLZ.TEXT
TIME_PACK	5.4.4.1.	106		TIME_PACK.TEXT
TRACK_AIDS	5.2.3.3.	62		TRACK_AIDS.TEXT
TRACK_PACK	5.2.3.3.	62	NEW_TRM	NEW_TRM.TEXT
TRIG_FUNC	5.4.2.3.	105		TRIG_FUNC.TEXT
TRM_PACK	5.2.3.1.	58	NEW_TRM	NEW_TRM.TEXT
TRM_TYPES	5.2.3.1.	58		TRM_TYPES.TEXT

Table 5-1. Utility Packages (Cont'd)

NOTE ON FILE STRUCTURE:

Each package is stored in a file of the same name. Due to a bug in the TeleSoft Compiler which only allows approximately 31 modules to be grouped into a single executive run, it wa necessary for us to group the packages into larger packages. The membership of the elemental packages in one of these larger packages is indicted in the third column of Table 5-1. An empty entry in this column means that the package was not concatenated into a super package.

The three file types are not independent of one another, but are for various purposes linked together to form larger structures.

The routines used to handle the three file types are found in separate packages as follows:

FILE TYPE	Ada PACKAGE	FILE NAME
Threat Tracking Files	THREATFILE	thrtfile.text
Threat Correlation Files	CORRELFILE	corrfile.text
Prioritized Threat List	PRIOTHLIST	priolist.text

There is, in addition, a fourth package named CREC_HANDLER (file name: crec_hdlr.text) which handles a subsidiary file type, the correlated item record (CREC), which is a subsection of a TCF record.

Many of the routines found in the three principal packages are carbon copies of each other, and indeed, would have been coded using Ada generics had that ability been available in the compiler used in this phase of the project.

Paragraph 1 introduces the doubly-linked ring structure used to implement each of the three principal file types. This paragraph also discusses the initialization of the reservoir of available file blocks maintained for each of the three file types. Paragraph 2 discusses the larger structures built up out of the basic three file types, illustrating these larger structures graphically and explaining their functional significance.

Paragraph 3 reviews the component fields of each of the three file types, explaining each component's function and giving an excerpt from the TRM code illustrating a typical usage of the given component field. Paragraph 4 deals with the four file handler packages listed above, emphasizing both the commonality of many of the procedures/functions provided and their significant differences. Paragraph 5 discusses lessons learned about Ada during Phase II with respect to the software covered by this section.

5.2.1.1. Doubly-Linked Ring Structures.

a. Basic Structure and Rationale. Each of the three principal file types is maintained as a doubly-linked ring. The salient features of such a structure as shown in Figure 5-3 are:

- The file as a whole consists of a number of (file) blocks, areas of contiguous memory that are, for a given file type, of uniform length;
- Subfiles are created by linking file blocks whose internal fields can be used to adduce some sort of useful association, for example, all file blocks that pertain to a particular sensor, stored in ascending azimuth order;

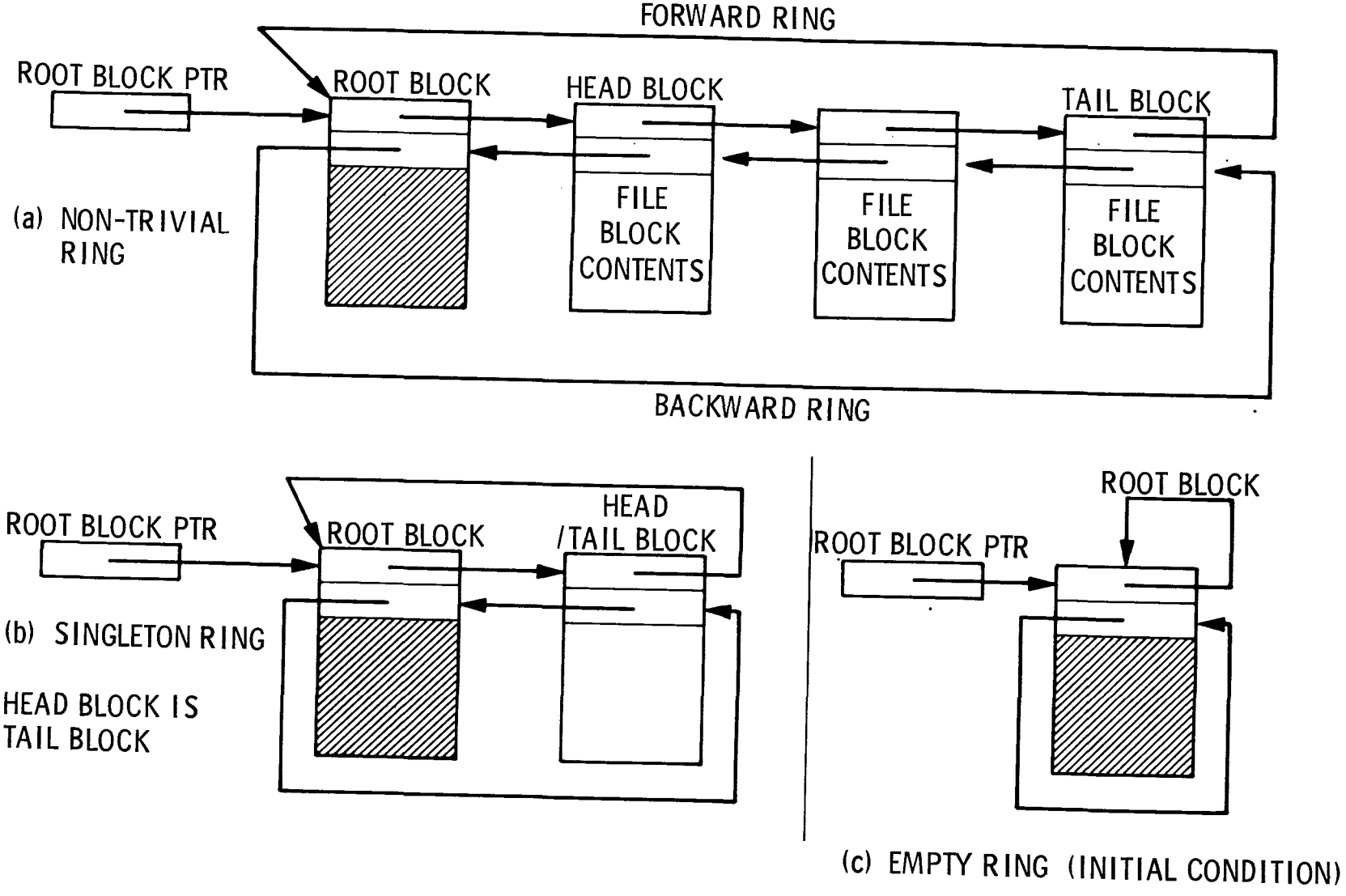


Figure 5-3. Doubly-Linked Ring Structures

- Each block contains a pair of pointers, a forward pointer and a backward pointer. The forward pointer within a given block points to its successor, and its backward pointer points to its predecessor;

- Three blocks are distinguished; these are the ROOT block, the HEAD block and the TAIL block. The root block is allocated once at the beginning of program execution, during the elaboration of a given file type's handler package, and is never allowed to disappear. The head block is the successor of the root, and the tail block is the predecessor of the root.

Having given this brief summary of the doubly-linked ring, we need to correct some over-simplifications and fill in some detail. Actually, as shown in Figure 5-4, a block contains an indexable list (array) of forward/backward pointer pairs. There is one such pair for each useful association required for a given file type. The orderings chosen in the present implementation are:

TTF: Ascending azimuth order within a sensor type;
Descending lethality order over all sensor types.

TCF: Descending lethality order over all sensor types.

PTL: Time of arrival order over all sensor types;
Descending lethality order over all sensor types.

The time of arrival ordering of the PTL means that the most recent PTL is in the head block and the oldest in the tail block. In all other orderings, ascending/descending refers to the forward direction around the ring.

Each of the orderings shown requires a root block. Accordingly, the PTL has two root blocks, the TCF has one and the TTF has seven: one for the lethality ordering and one for each of the six sensor types (see Figure 5-5). These root blocks are not referred to directly, but as shown in both Figures 5-3 and 5-5, each root block is accessed via its root block pointer which is an Ada access type object.

Further, the root blocks are an implementation detail that are of no concern to the TRM user routines. The usual practice is to first verify that the desired ordering is not empty (an empty ordering consists of a root block pointing to itself --(see Figure 5-3), and then obtain pointers to the ordering's head and tail blocks. These pointers are the only useful part of a root block; no other information is stored in the fields of a root block. This is somewhat wasteful of space, in that assembly language versions of doubly-linked rings define the equivalent of the root block as consisting only of the pointers, but it is required by Ada's strong typing: all the pointers for a given file type are of the same Ada access type, namely pointers to the file blocks of the file type. A root block consisting only of pointers would not be of the same type as the usual file block and thus the forward pointer of the tail block and the backward pointer of the head block could not point to it, without the messy inconvenience of variant records or the subterfuge of unchecked conversion.

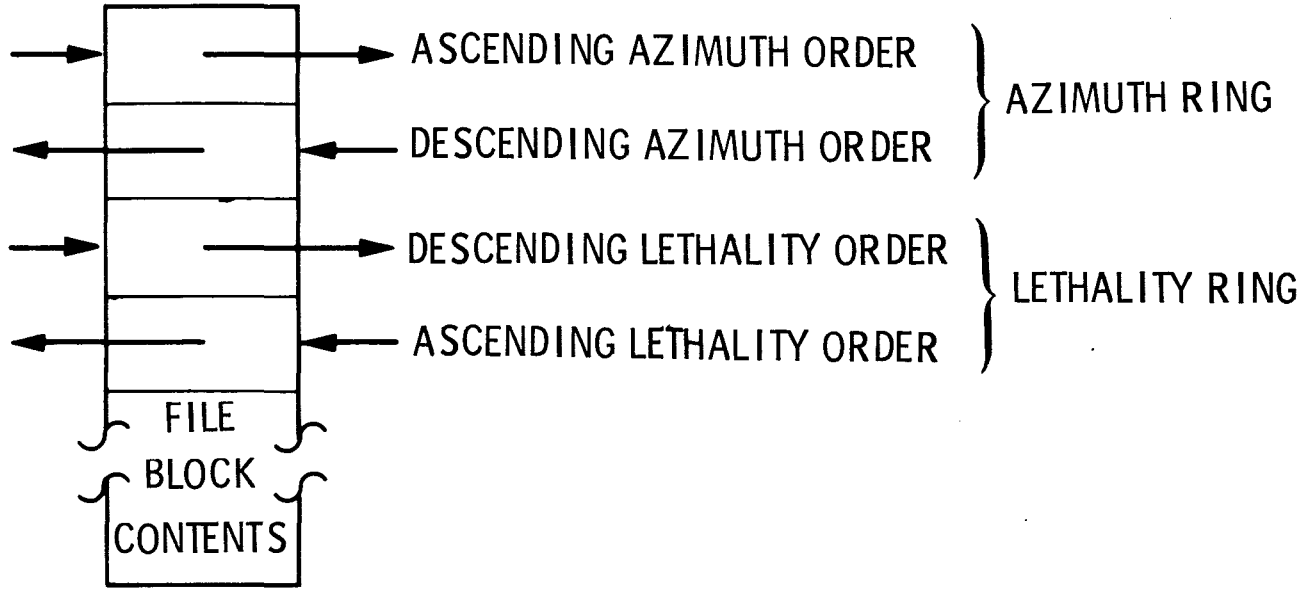


Figure 5-4. Simultaneous Ordering of a Single Block. Example Shown Pertains to TTF.

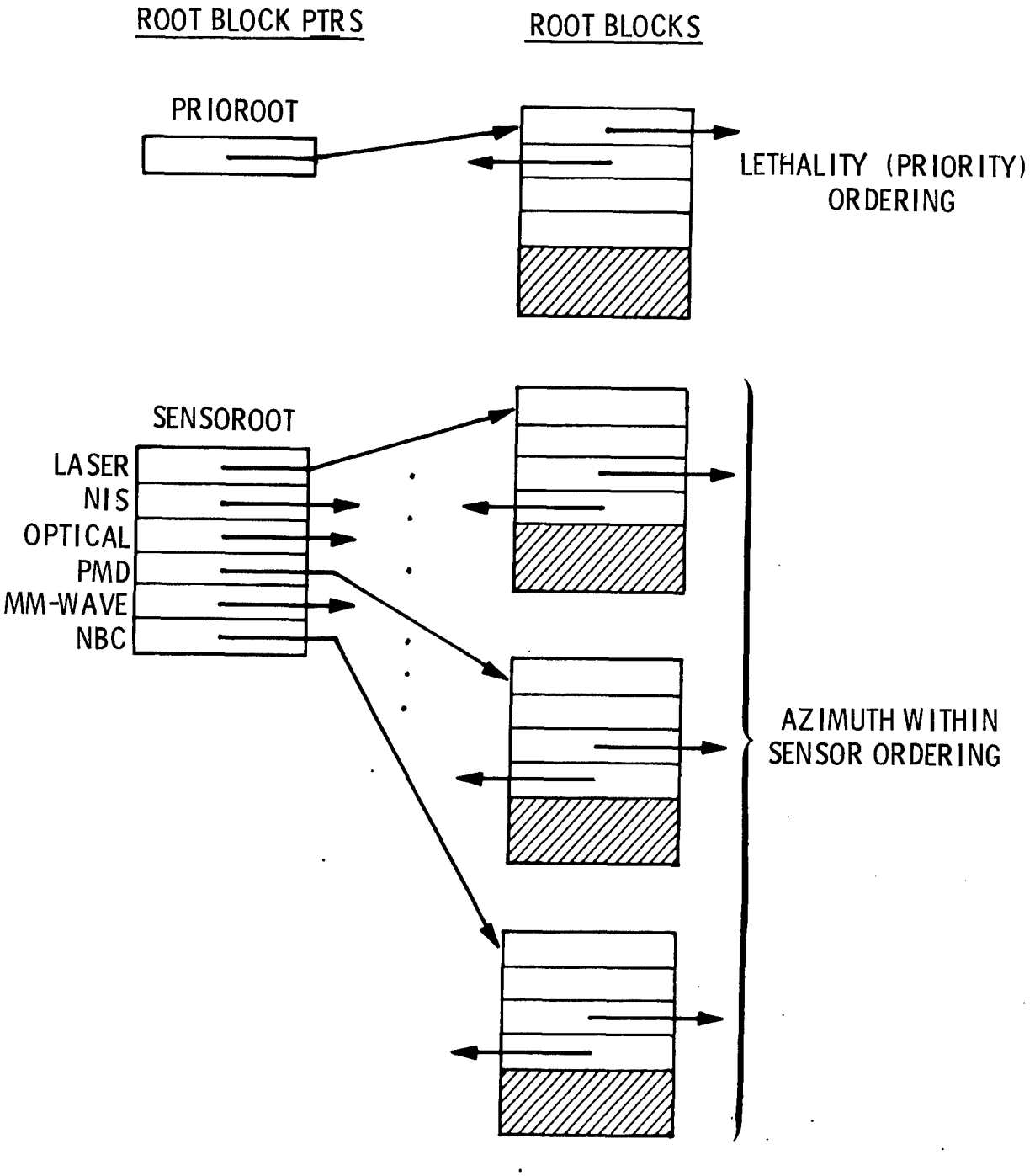


Figure 5-5. A Root Block for Each Ordering. Example Shown Pertains to TTF.

This design choice, i.e., not allowing the root blocks to introduce an inconvenient exception, is quite in keeping with the rationale for choosing the doubly-linked ring structure in the first place. Maintaining a file in some specific order means that a block will sometimes have to be severed from its current position and moved to a different position. The doubly-linked ring permits this to be done with minimal exceptions: the rules for removing a block from a ring are the same no matter where (head, tail, or middle) that block is and without regard to whether or not the ring has only one block on it (is a "singleton"). Similarly, the rules for adding a new block to a ring are the same for an empty ring or a singleton and without regard for position in a non-trivial ring. The sole exception is that nothing may be removed from an already empty ring, i.e., the root block will be protected.

By way of comparison, the correlated item record handler (CREC_HANDLER) does many of the same functions as the three principal handlers. The file structure deemed apropriate here was not the doubly-linked ring, but a doubly-linked chain structure in which the forward pointer in the tail block points to nothing (is null) and the same for the backward pointer in the head block. This is illustrated in Figure 5-6. The FREE_ONE function in CREC_HANDLER which corresponds to the DETATCH function of the other three handlers recognizes five separate cases: chain already empty, detaching a singleton, detaching chain head, detaching chain tail and detaching a middle item. DETATCH recognizes only two cases: ring empty and all other situations; the code required to implement all other situations consists of essentially the same two lines that FREE_ONE uses for its last case alone.

b. Initialization. A full-capability, validated Ada compiler provides facilities for dynamic allocation and deallocation of storge. Thus, during the elaboration (execution of initialization code) of package THREATFILE, having declared PRIOROOT to be an access variable (pointer) to a TTF_REC (file block for TTF), i.e., of type TTF_ PTR, we use the Ada allocator 'new' to create the permanent root block for the TTF priority ordering with the following code statement:

```
PRIOROOT: = new TTF_REC;
```

The key word here is "permanent." The root blocks persist throughout the TRM's execution and are never deallocated. By contrast, the ordinary file blocks making up the bulk of the dynamic files must be OBTAINed when needed and DELETEd when no longer needed. Since the developmental version of the compiler that was used in developing the Phase II software has the facility for dynamic allocation illustrated above, but does not have the corresponding facility for dynamic deallocation, we were compelled to provide our own functional allocation and deallocation scheme. This scheme, which is used in the three principal handlers and in the CREC_HANDLER, is illustrated in Figure 5-7 through 5-11 for the TTF blocks created and handled by package THREATFILE.

The gist of our allocation/deallocation scheme is as follows: during elaboration of all four packages, a fixed number of file blocks is created

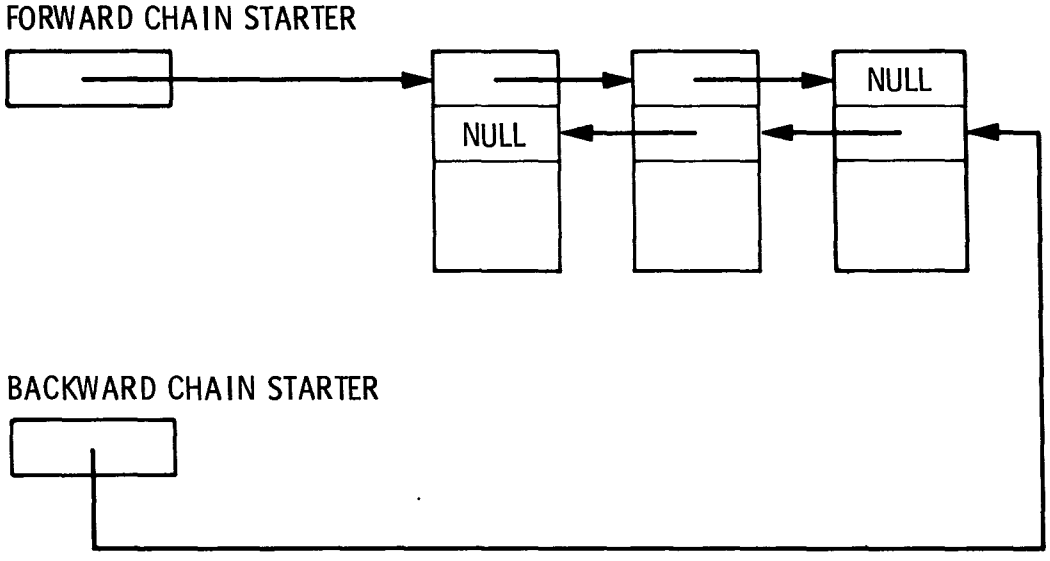
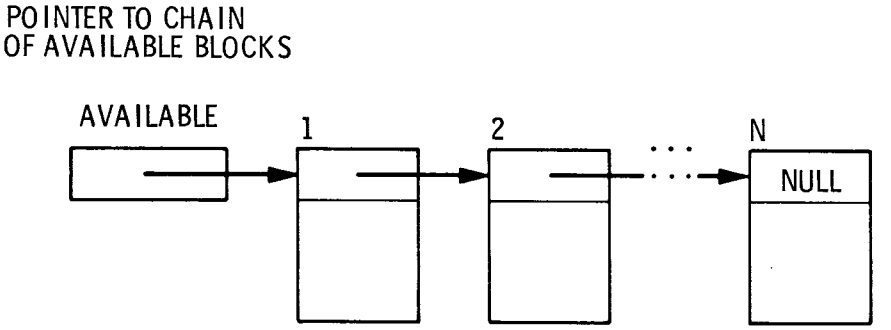


Figure 5-6. Doubly-Linked Chain Structure (CREC_HANDLER)



(a) INITIAL RESERVOIR OF AVAILABLE BLOCKS

FOR	THREATFILE	:	N = POOL_SIZE
	CORRELFILE	:	N = POOL_SIZE/2
	PRIOTHLIST	:	N = POOL_SIZE
	CREC_HANDLER	:	N = POOL_SIZE

Figure 5-7. Allocation/Deallocation of File Blocks. (Initial Reservoir of Available Blocks)

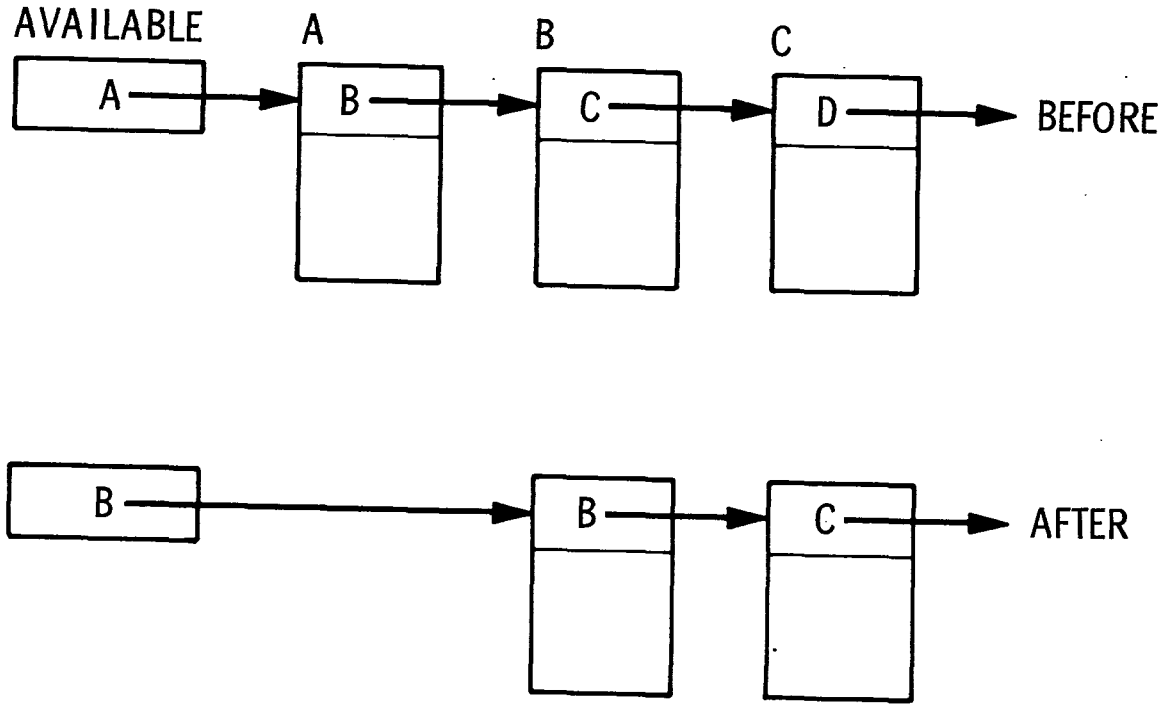


Figure 5-8. Allocation/Deallocation of File Blocks (Obtaining an Available Block) (Cont'd)

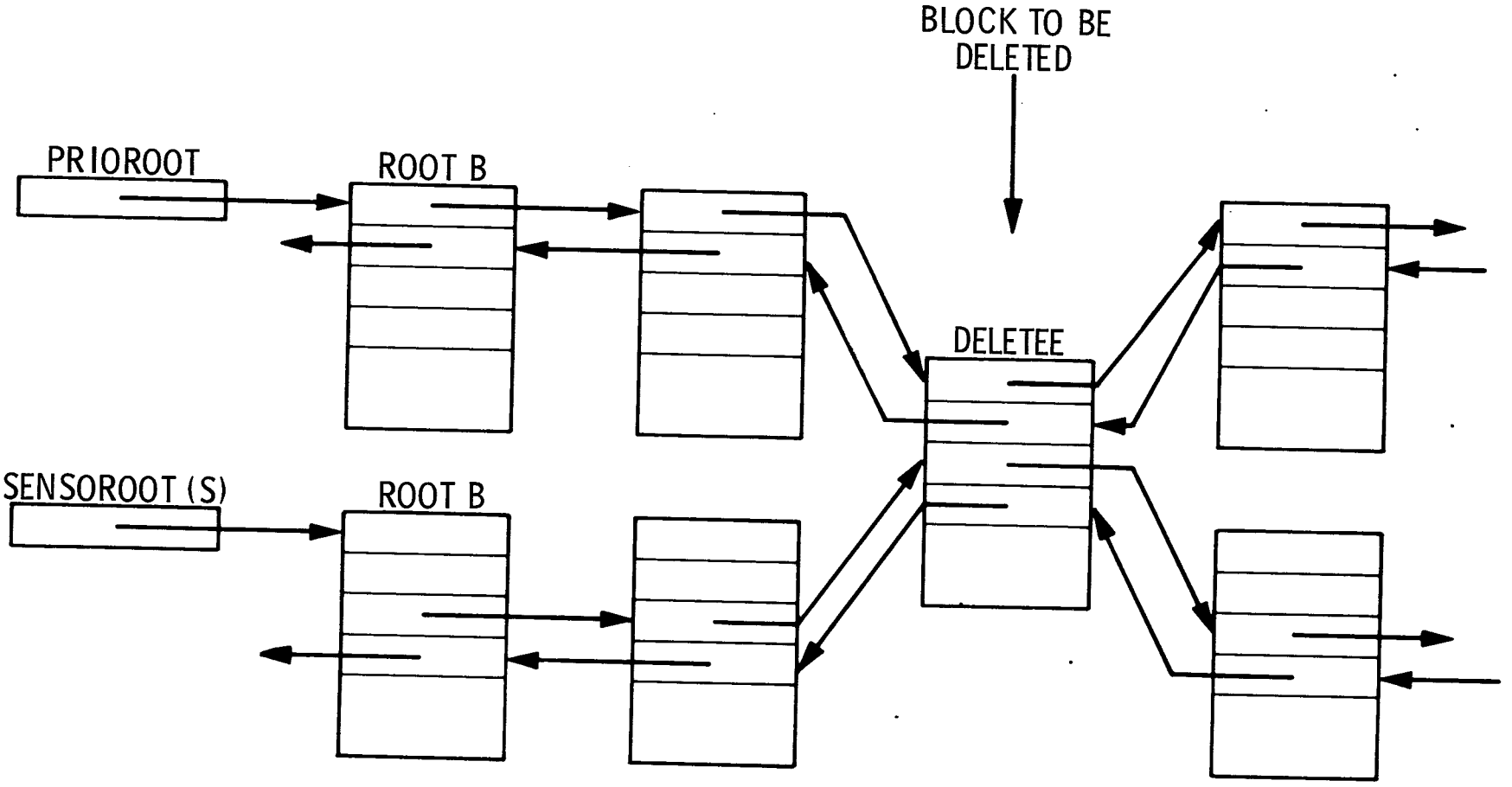


Figure 5-9. Allocation/Deallocation of File Blocks (Deleting a Block: Initial State) (Cont'd)

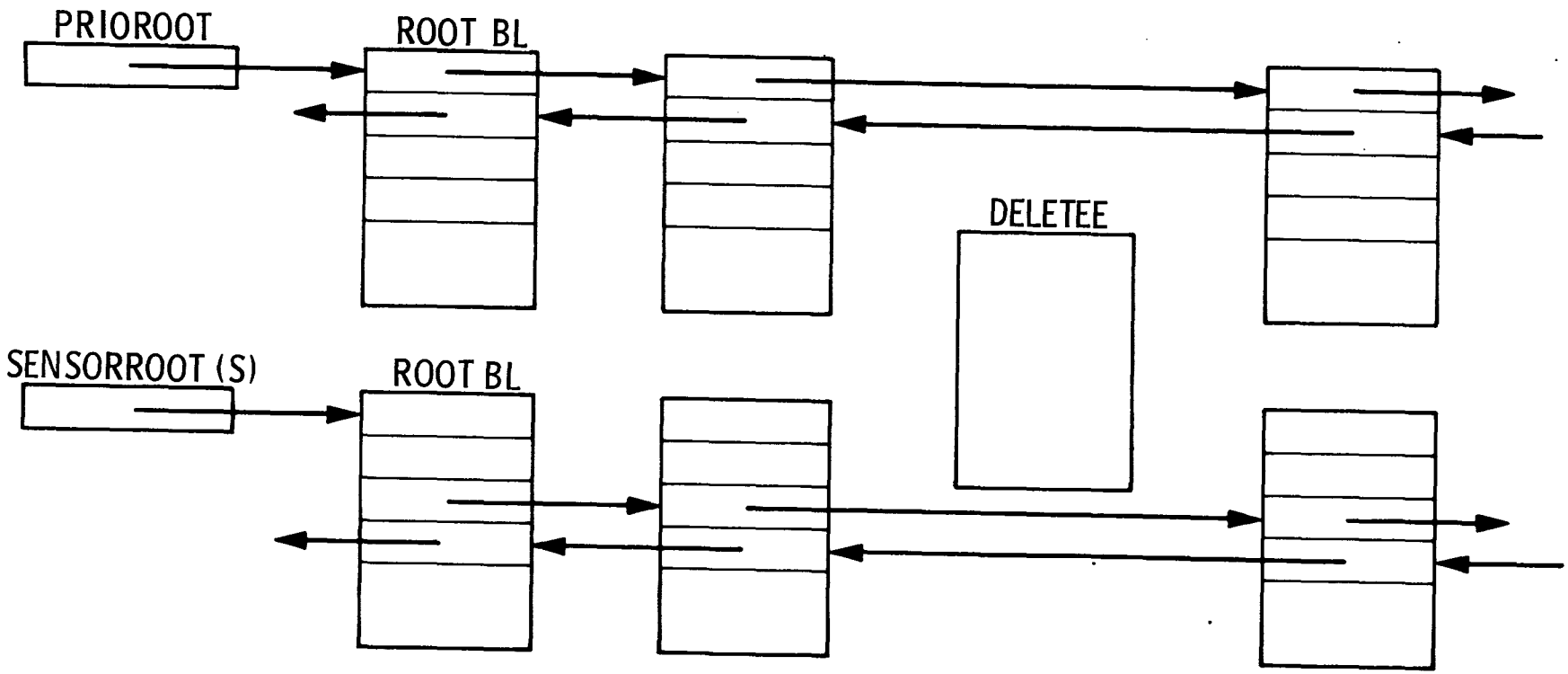


Figure 5-10. Allocation/Deallocation of File Blocks. (Deleting a Block: After Severing Ring Connections) (Cont'd)

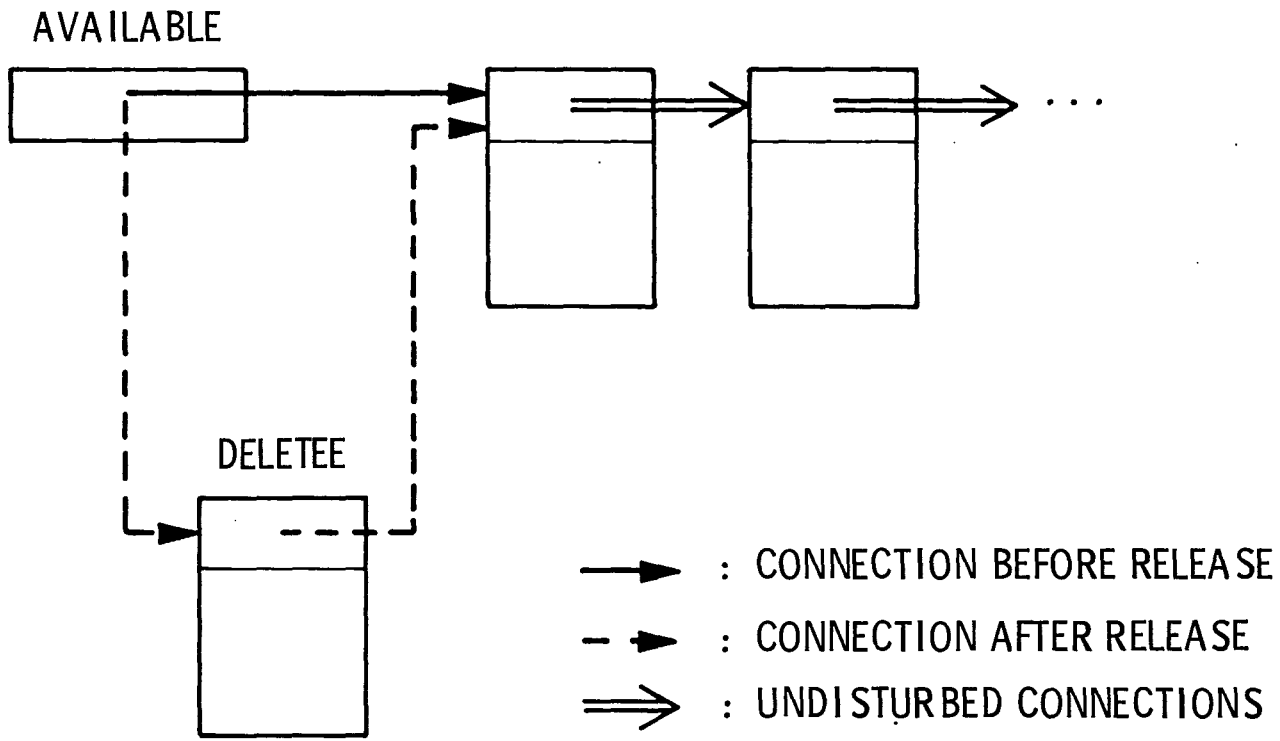


Figure 5-11. Allocation/Deallocation of File Blocks. Deleted Block Release to Reservoir of Available Blocks (Cont'd)

using the Ada allocator 'new.' These blocks are hung on a unidirectional chain whose chain_starter is named 'AVAILABLE' via a code segment such as the following taken from CREC_ HANDLER:

```
--AVAILABLE is initially null
--TEMP, AVAILABLE and the NEXT field of a COR_REC are
--all of type COR_PTR which is access COR_REC

for I in 1..THREATFILE.POOL_SIZE loop
     TEMP            : = AVAILABLE;
     AVAILABLE       : = new TRM_TYPES.COR_REC;
     AVAILABLE.NEXT  : = TEMP;
end loop;
```

The result of this loop's execution is shown in Figure 5-7.

To allocate a file block during TRM execution, one calls a procedure named OBTAIN (Figure 5-8) which returns the current value of AVAILABLE (pointer to the next available unused block if not = null. This value will be null if the last previous call on OBTAIN (for this block type) used the last available block; the user must interpret this null return as an indication that there are no more blocks available-see remark below. If this situation arises, then OBTAIN does nothing further, otherwise AVAILABLE is reset to the contents of the available_ chain_pointer field of the block (e.g., the NEXT field in the above example). Since a TCF block is OBTAINed when a fresh correlation has been discovered between two TTFs, procedure CORRELFILE.OBTAIN also calls CREC_HANDLER.OBTAIN to obtain two COR_REC blocks to represent the two TTFs being correlated. These COR_REC blocks are attached to the chain starters which are contained in the TCF block being OBTAINed (see Figure 5-7).

To deallocate a file block during TRM execution, one calls a procedure named DELETE. This procedure uses of other handler procedures to first DETATCH the block from its current ring connections, after which it will RELEASE the floating block to the reservoir of available blocks. These stages are shown in Figures 5-9 through 5-11. The RELEASE procedure also sets all fields to some predetermined value. Corresponding to the exception for the TCF noted above, procedure CORRELFILE.RELEASE calls procedure CREC_HANDLER.FREE_ALL to unchain and release all COR_RECs currently attached to the TCF block being RELEASEd.

REMARK - Concerning running out of file space, note from Figure 5-7 that all handlers base the number of blocks allocated on the same constant, POOL_SIZE, which is defined in and exported to the other handlers from package THREATFILE. The one-to-one correspondence between the number of TTF blocks and the number of COR_RECs is obvious. The respective numbers of TCF and PTL blocks are based on worst-case analysis: Since each TCF block corre-lates two or more TTF blocks, the maximum number of TCF blocks required would be one-half the number of TTF blocks. On the other hand,

Figure 5-12 (next) shows that if every TTF aged_in without being correlated, then the maximum number of PTL blocks required would be the same as the number of TTF blocks. The net result of this linkage of pool sizes is that the only OBTAIN procedure requiring critical attention to running out of room is THREATFILE.OBTAIN. This event, when it occurs, is detected in procedure TRACK_PACK.TRACK when a call on TRACK_AIDS.MATCH determines that the current sensor input represents a "NEWGUY" for which a new TTF entry must be established. When the subsequent call on THREATFILE.OBTAIN fails (returns with a null pointer), procedure TRACK_ AIDS.FIND_ROOM is called to see if it is possible to eliminate a TTF with a smaller estimated lethality than that of the current input (an over-simplified summary). This quest may or may not succeed; if it does, then the less threatening TTF is DELETEd and the current input claims its space, otherwise, the current input is passed over, i.e., denied entry to the TTF.

5.2.1.2. File Super-Structures. In addition to the ring pointers discussed above, each of the three principal file type blocks contains pointers to the other two types. These enable TTFs, TCFs, and PTLs to be joined together into super-structures for various purposes. Each link in such a super-structure is bidirectional, so for example, if a TTF is part of a correlated object, then it points to the TCF which represents that correlated object while the TCF contains as many pointers as necessary to point to all the TTFs which it correlates together.

A TTF represents the observation of a "something out there" by one particular sensor, having sensor-measured parameter values sufficiently separated from those of other observations by the same sensor to warrant the establishment of a new TTF. When first established, such a TTF is neither correlated nor aged_in. This is manifested by having its correlation_pointer (TTCFP) and its aged_in_pointer (TPTLP) both set to null.

Correlation is the process of discovering that two different sensors have detected an object that is at the same physical location (within reasonable error windows related to the accuracy with which the sensors involved measure the parameters that specify physical location). Once such a positive finding has been established, the TTFs involved are said to be correlated or members of a correlated object.

Independently of correlation, our TTF might be given a lethality assessment that is sufficiently high, or be detected sufficiently many times as to merit being aged_in, i.e., considered worthy of display to the crew or of some other sort of reaction/countermeasure response. When this happens, a PTL file block is established for the object represented by the qualifying TTF.

Returning to our newly established TTF, in its subsequent history, one of five things may happen:

1. It may remain both uncorrelated and not aged_in;

2. It may become correlated, but not aged_in;

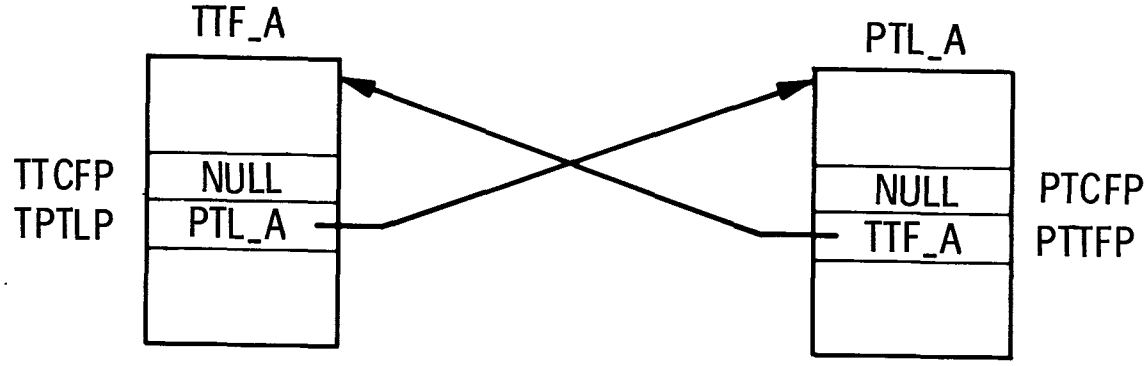


Figure 5-12. Aged_In, Not Correlated

3. It may become aged_in, but not correlated;

4. It may become aged_in and then correlated;

5. It may become correlated and then aged_in.

Situation 1 is trivial and uninteresting. Situation 2, correlated but not aged_in, represents the first of our super-structures and is illustrated in Figure 5-13. (In this figure and the others dealing with related super-structures, we concentrate on the interconnections between the TTF, TCF and PTL block types and suppress the details of the individual block types' internal ring connections). As noted above, each TTF has non-ring pointer fields TTCFP and TPTLP for pointing to associated TCF and PTL blocks. In situation 2, pointer TTCFP points to the TCF representing the correlation object to which the TTF belongs; the TPTLP pointer remains null. On the TCF side of this super-structure, we represent the TTFs "owned" by the TCF as a doubly-linked chain of COR_RECs as discussed in a previous section. The chain-starters for this chain are called COR_FIRST (forward) and COR_LAST (backward). Each COR_ REC consists of a forward pointer (NEXT), a backward pointer (PREV), and a pointer to a TTF (COR_ITEM).

Situation 3, aged_in but not correlated, is illustrated in Figure 5-12. The TTF's non-ring pointers are set so that TTCFP is now null while TPTLP points to the PTL block. On the PTL side of this super-structure, there are two non-ring pointers, PTTFP which points to the aged_in TTF and PTCFP which is null.

Situations 4 and 5 both end up with the super-structure shown in Figure 5-14. Once correlation occurs, we no longer regard the TTFs as being aged_in; it is the correlated object which is aged_in. Thus, direct connections between TTFs and PTLs which might have existed previously are severed and the aged_in state of the correlated object is represented by a TCF field, CPTLP which points to the appropriate PTL. On the PTL side of this relationship, PTL pointer field PTCFP now points to the aged_in TCF. In situation 4, if two aged_in TTFs become correlated, only one PTL block survives.

The net result of the approach outlined in the previous paragraph is that in an active TTF, the pointers TTCFP and TPTLP cannot be non-null simultaneously. On the other hand, exactly one of the pointers PTCFP and PTTFP must be non-null in an active PTL.

5.2.1.3 File Block Component Fields.

a. TTF File Block Components. A TTF file block is an Ada record of type TTF_REC. Its access type, TTF_PTR, is the type of all the ring pointers used, as discussed above, to create the TTF orderings. It is also the type of the TTF root block pointers, the pointers COR_ITEM attached to the TCF file block via COR_FIRST/LAST, and the pointer PTTFP of the PTL file block. These types as well as the types of almost all of the other components of the TTF file block are defined in and exported from package TRM_TYPES.

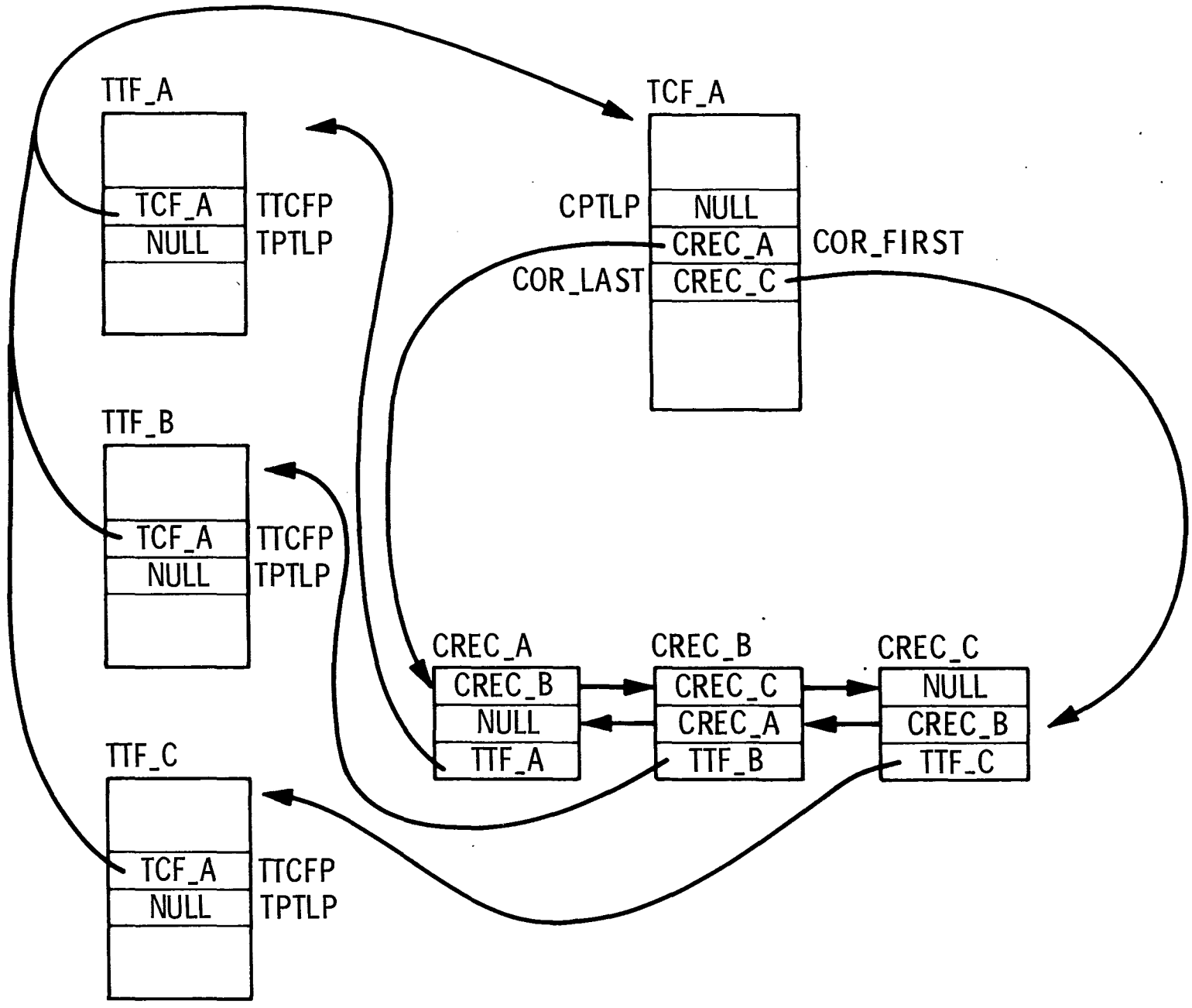


Figure 5-13. Correlated Without Age_In

The TTF block is divided into three major sections:

- o The pointer section;
- o The parameter section;
- o The miscellaneous field section.

The pointer section includes the ring pointers and the pointers TTCFP and TPTLP whose functional significance has been discussed above. The ring pointers are organized into two arrays, TFRWD and TBKWD for the forward and backward ring pointers, respectively. Each of these arrays is indexed by a variable of type TTF_RANK which can take one of the two values:

TTF_SENSOR_AZM for the azimuth within sensor ordering and
TTF-GLOBAL_PRIORITY for the priority/lethality ordering.

These cumbersome literals are often aliased with more convenient names such as ANGLE for the first and PRIORITY for the second by use of the Ada "renames" declaration. Assuming this aliasing has already been done, the following condensed example taken from procedure MATCH in package TRACK_AIDS (file name: trak_aids.text) shows a typical use of the ring pointers to inspect all the file blocks on a particular ring.

```
--S_PTR is set initially to point at the head block of the azimuth-
  ordered ring for sensor SENSOR;
--S_LAST is set initially to point at the tail block of the same
  ring;
loop
--Perform match tests on the parameters of the file
--block pointed to by S_PTR;
  exit when S_PTR = S_LAST;
  S_PTR := S_PTR.TFRWD(ANGLE); -- Move to next block
end loop;
```

Typical uses of the pointers TTCFP and TPTLP involve differentiating an action depending on whether a TTF is correlated, aged_in or neither:

```
if      TRACK_FILE.TTCFP /= null          -- Correlated?
then    TCFIL := TRACK_FILE.TTCFP;
        ........
elsif   TRACK_FILE.TPTLP /= null          -- Aged_In?
then    PTLFL := TRACK_FILE.TPTLP;
        .......
else    .......                           -- Neither?
end     if;
```

The parameter section is a single field named SIPDATA. SIPDATA is a record of type SIP_RECORD defined in and exported from package SIP-PACK. The fields of SIPDATA are as follows:

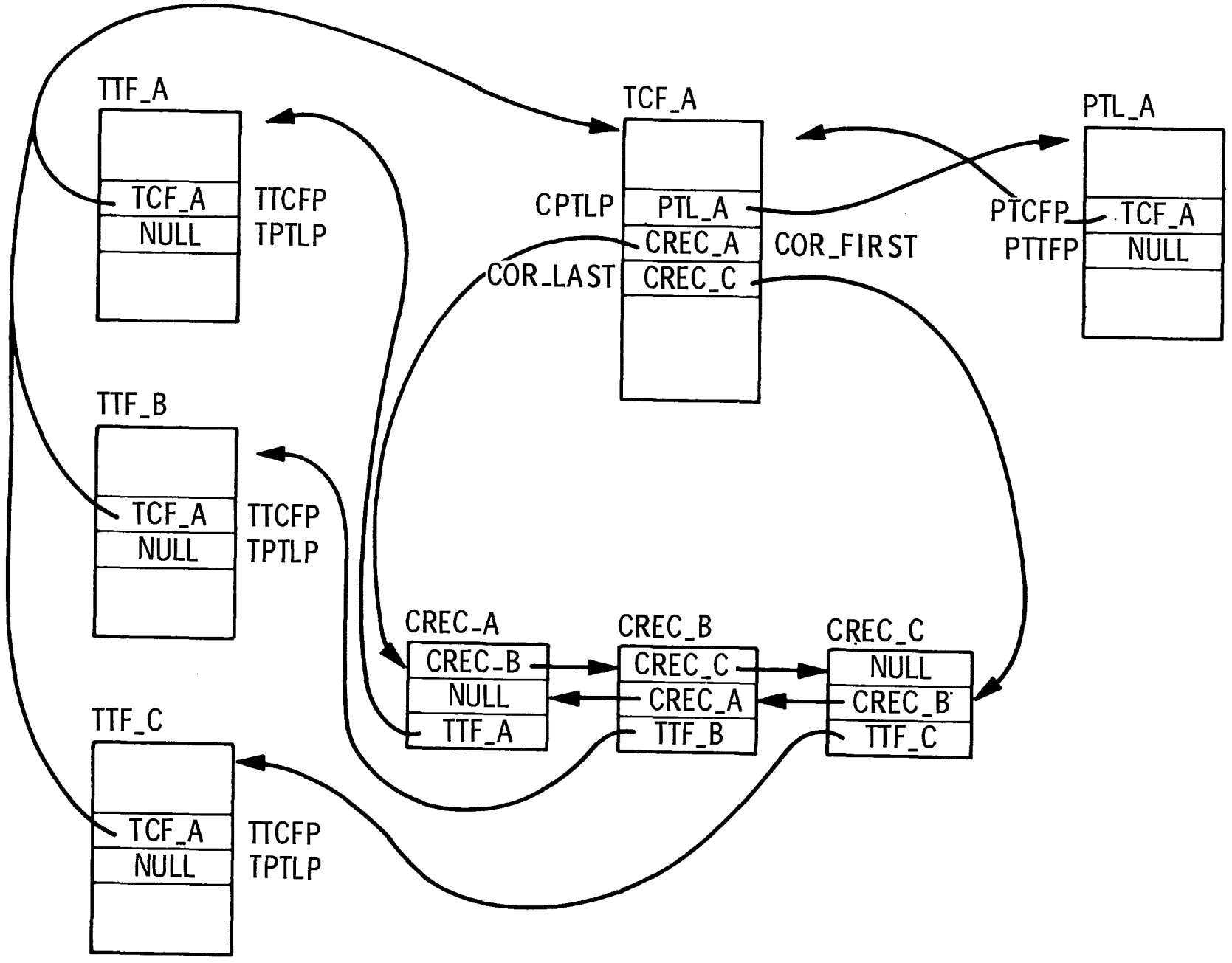


Figure 5-14. Correlated and Aged_In

```
SENSOR_ID       --Specifies the sensor:  LASER, NIS, etc.
EMITTER         --A sensor-specified subtype
RANGE_WORD      --Distance in meters if sensor measures range, other-
                  wise a dummy
AZIMUTH         --Compass angle in degrees
ELEVATION       --Zero degrees at horizon; 90 degrees at zenith
TIME_SENSED     --Provides clock time on input, but is never altered;
                  intended for throughput time measurements when
                  realtime clock capability becomes available.
```

This parameter data can be handled at the gross level of an entire SIP_RECORD or at the finer level of individual parameters. At the gross level, we find in procedure CREATE_TTF (package TRACK_AIDS, file trak-aids.text) the following statement:

```
TRACK_FILE.SIPDATA := THIS_SIP;
```

TRACK_FILE is a pointer to the TTF being created, THIS_SIP is a pointer to the SIP_RECORD with the data to be used; both are input arguments to the procedure.

A typical use of individual parameters is seen in procedure UPDATE_TTF (same package as CREATE_TTF) in which the TTF block pointed to by input argument TRACK_FILE is being updated by the more current information contained in input argument THIS_SIP:

```
--N is the number of times that the TTF block accessed by TRACK_FILE
  has previously been seen;

--NEW_AVERAGE (X, N, Y) ==> X := (N*X +Y) / (N+1);

if CAN_MEASURE(THIS_SIP.SENSOR_ID, RAYNJ)
THEN NEW_AVERAGE(TRACK_FILE.SIPDATA.RANGE_WORD,
                 N,        THIS_SIP.RANGE_WORD);
end if;
```

The miscellaneous fields section contains three fields:

```
TTOA    --Latest time the object represented by this block has been
          seen;

TPRIO   --The threat priority/lethality of this object;

AICNT   --Age_In_CouNT:  The number of times this object has been
          seen; used as N in the preceding example (after conversion
          to floating point).

EXAMPLE:  N := FLOAT (TRACK_FILE.AICNT); -- In UPDATE_TTF.
```

b. TCF File Block Components. A TCF file block is an Ada record of type TCF_REC. Its access type, TCF_PTR, is the type of all the ring pointers used, as discussed above, to create the single TCF ordering. It is also the type of the TCF root block pointer and the pointers TTCFP and PTCFP of the TTF and PTL file blocks, respectively. These types as well as the types of almost all of the other components of the TCF file block are defined in and exported from package TRM_ TYPES.

The TCF block is divided into three major sections:

- The pointer section;

- The history section;

- The miscellaneous fields section.

The pointer section includes the ring pointers, the pointers COR_FIRST, COR_LAST, and CPTLP whose functional significance has been discussed above. The ring pointers are organized into two arrays, CFRWD and CBKWD for the forward and backward ring pointers, respectively. Each of these arrays is indexed by a variable of type TCF_RANK which can take the single value

TCF_GLOBAL_PRIORITY for the priority/lethality ordering.

This cumbersome literal is often aliased with the more convenient name PRIORITY or PRIOR_C by use of the Ada "renames" declaration. Assuming this aliasing has already been done, the following condensed example taken from procedure FIND_ROOM in package TRACK_AIDS (file name: trak_aids.text) shows a typical use of the ring pointers to inspect all the file blocks in the priority ring.

```
--TCFL        is set to point to the tail block of the priority ring;
--LAST_TCFL   is set to point to the head block, same ring;
--CPRIO       is explained later in this section;
loop
  exit when TCFL.CPRIO >= LETHALITY;

  if  TCFL.CPTLP /= null
  then PTLIT := TCFL.CPTLP;

  else goto  MAKE_ROOM;
  end if;

  exit when TCFL = LAST_TCFL;

  TCFL := TCFL.CBKWD(PRIOR_C);  -- Move to previous TCF
end loop;
```

The history section consists of two simple and one complex fields; these are:

HISCOUNT --A count of the number of histories saved thus far; can range from 0 to NHIST, a constant currently fixed at 5;

YOUNGEST --The index into the array of histories which locates the most recent one; can range from 0 to NHIST-1;

SIGHTING --The array of histories with a maximum number elements = NHIST.

Each element of SIGHTING is a record of type HISTO_REC whose four fields are:

WHO --A pointer to a TTF block;

RNGE --Distance in meters;

AZIM --Compass angle in degrees;

TYME --The time of the sighting.

SIGHTING is a circular array in that once its capacity is reached, each new entry is stored over the currently oldest entry. Not every TTF that correlates into a TCF is entered into the history array; the sensor associated with the TTF must be able to measure range, i.e.:

CAN_MEASURE(XXXX.SIPDATA.SENSOR_ID, RAYNJ)

must be true. It is assumed that range sensors measure azimuth, but not vice-versa.

The following example of the use of the history section is taken from TRACK_AIDS.ANALYZE_MOTION:

```
--LATEST      is of type HISTO_REC;
--TCFIL       points to a TCF block;
--THIS_SIP    is a SIP_RECORD input argument to the procedure;
LATEST     := TCFIL.SIGHTING(TCFIL.YOUNGEST);
TF_RANGE   := LATEST.RNGE;
DEL_T      := THIS_SIP.TIME_SENSED - LATEST.TYME,
DEL_AZ     := THIS_SIP.AZIMUTH     - LATEST.AZIM;
```

History handling procedures/functions are provided by package TRM_TYPES; these are function NEXT_HINDEX (moves the history array index circularly forward), function PREV_HINDEX (moves the history array index circularly backward) and procedure SAVE_HISTORY.

The miscellaneous fields of a TCF block are:

RANGERS --Integer count of the associated TTFs that measure range; a positive value answers affirmatively the question: can this correlated object measure range?

GUIDERS --Integer count of the associated TTFs that illuminate a target; used by procedure AGE_IN_PAK.TEST_WARNINGS to raise an ILLUMINATED warning on a hovering object;

IS_PLAT --A boolean flag that says whether or not this correlated object is a platform; at present this is equivalent to asking whether or not there is a NIS sensor among the TCF's associated TTFs.

CTOA --This is the latest time seen of any of the associated TTFs on the doubly-linked chain located via COR_FIRST and COR_LAST; this time may differ from the time of the most recent SIGHTING, because of the range-measurement requirement on a sighting.

CPRIO --This is the maximum TPRIO over the chain of associated TTFs.

Example: See previous example from TRACK_AIDS.FIND_ROOM.

c. PTL File Block Components. A PTL file block is an Ada record of type PTL_REC. Its access type, PTL_PTR, is the type of all the ring pointers used, as discussed above, to create the PTL orderings. It is also the type of the PTL root block pointers and the pointers TPTLP and CPTLP of the TTF and TCF file blocks, respectively. These types as well as the types of almost all of the other components of the PTL file block are defined in and exported from package TRM_ TYPES.

The PTL block is divided into three major sections:

- The pointer section;

- The flags section;

- The miscellaneous fields section.

The pointer section includes the ring pointers and the pointers PTCFP and TTTFP whose functional significance has been discussed above. The ring pointers are organized into two arrays, PFRWD and PBKWD for the forward and backward ring pointers, respectively. Each of these arrays is indexed by a variable of type PTL_RANK which can take one of the two values:

PTL_GLOBAL_TOA for the time-of-arrival ordering and
PTL_GLOBAL_PRIORITY for the priority/lethality ordering.

These cumbersome literals are often aliased with more convenient names such as T_OF_ARR for the first and PRIORITY for the second by use of the Ada "rename" declaration. Assuming this aliasing has already been done, the following condensed example taken from procedure AGO_PACK.AGE_OUT shows a typical use of the ring pointers to inspect all the file blocks on a particular ring.

```
      --THIS_ITEM is set to point to the oldest PTL block on the
                  T_OF_ARR ring;

      --LAST_ITEM is set to point to the youngest such PTL block;

      --PTOA      is explained later in this section;

   loop

      AGE    := CLOCK_TIME - THIS_ITEM.PTOA;

      exit when AGE < MIN_AGE_OUT_TIME;

      NEXT_ITEM := THIS_ITEM.PBKWD(T_OF_ARR);

      if    THIS_ITEM.PTCFP /= null

      then  -- Complex logic which may result in the deletion
            -- of all the TTFs attched to THIS_ITEM.PTCFP,
            -- the TCF block pointed to by THIS_ITEM.PTCFP,
            -- and the PTL block pointed to by THIS_ITEM itself.

      else  -- Less complex logic which may result in the
            -- deletion of the TTF block pointed to by
            -- THIS_ITEM.PTTFP and the PTL block pointed to
            -- by THIS_ITEM itself.

      end if;

      exit when THIS_ITEM = LAST_ITEM;

      THIS_ITEM := NEXT_ITEM;

   end loop;
```

The flag section contains two flag fields; these are:

```
   STATUS   --An enumeration value taking one of the following values:

            NONE          --Not yet seen by REACTION_DECISION
            WAITING       --Awaiting reaction
            ACTIVE        --Reaction in process
            COMPLETED     --Reaction has been completed
```

```
PFLAG     --A warning flag taking on one of the following self-explana-
            tory values:

            NULL_WARNING
            ILLUMINATED
            HIND_CLOSING
            SCOUT_NEARBY
            UNIDENTIFIED_OBJECT_CLOSING
```

The elided logic in the previously given example from TRACK_AIDS.FIND_ROOM calls a function P_TEST:

```
--PRIO is explaned later in this section;

function P_TEST (P_FILE : in TRM_TYPES.PTL_PTR) return BOOLEAN is
begin
return P_FILE.STATUS  = COMPLETED or else
       (P_FILE.STATUS /= ACTIVE   and then
        P-FILE.PPRIO < LETHALITY);
end P-TEST;
```

The miscellaneous fields section contains the following:

```
PTOA    --Let the pointer to the PTL block be denoted by PTLIT. If
          PTLIT.PTCFP /= null, then PTOA has the same value as
          PTLIT.PTCFP.CTOA, otherwise, PTOA has the same value as
          PTLIT.PTTFP.TTOA

PPRIO   --Let the pointer to the PTL block be denoted by PTLIT.  If
          PTLIT.PTCFP /= null, then PPRIO has the same value as
          PTLIT.PTCFP.CPRIO, otherwise PPRIO has the same value as
          PTLIT.PTTFP.TPRIO
```

Examples have been given earlier in this section of both PTOA and PPRIO.

5.2.1.4. File Handler Routines. As noted earlier, each of the three principal file types, TTF, TCF, and PTL, is manipulated by its own package of functions and procedures. Most of these functions and procedures are practically identical from package to package except for the type of file block handled, and could have been used via Ada generics if they had been available. In the first part of this section, we will discuss these routines generically, pointing out the few places where they differ. In the second part of this section we will discuss the functions and procedures provided by package CREC_HANDLER.

a. The Doubly-Linked Ring Handlers. Each of the packages that provides facilities for handling doubly-linked rings (THREATFILE, CORREL-FILE, PRIOTHLIST) provides the following functions and procedures:

OBTAIN --A procedure for getting a pointer to a free file block. This procedure has been discussed and illustrated at some length in a previous secton. As also noted in this same section, procedure CORRELFILE.OBTAIN also calls procedures CREC_HANDLER.OBTAIN and INSERT twice each to get and attach two COR_RECs to the COR_FIRST/LAST pointers of the TCF file block just obtained.

RELEASE --A procedure for putting an unneeded file block back into the reseervoir of available file blocks, and for clearing out all its fields to some predetermined set of values. This procedure was also discussed and illustrated at some length in a previous section. As also noted in this same section, procedure CORRELFILE.RELEASE first calls procedure CREC_HANDLER.FREE_ALL to detach and make available all COR_ RECs attached to the TCF file block being released.

DETATCH --There are two overlayed DETATCH procedures for each file block type. In the first of these, one supplies a pointer to the file block to be detatched and another argument of type TTF_ RANK, TCF_RANK, or PTL_RANK which specifies which of the rankings is intended. Thus, for example, when the priority field of a file block changes, one detatches that block from its priority ring before RE_PRIORITIZING it. This first form of DETATCH affects the pointers of the specified ring only, leaving the other rings (if any) intact. The second form of DETATCH is called with the second argument literally equal to "ALL_LINKS", and it calls on the first form of DETATCH to sever the block from all of its ring attachments. This second form of DETATCH is called by DELETE, as discussed in a previous section.

DELETE --DELETE combines the actions of the second form of DETATCH and RELEASE, in that order, to sever all ring attachments of a block and return it to the reservoir of available file blocks. This procedure was illustrated and discussed at some length in a previous section.

INSERT --INSERT is called with four arguments: (1) The INSERTEE = a pointer to the block which is being inserted; (2) The RING = PRIORITY, ANGLE, T_OF_ARR on which the insertion is to take place; (3) BEFORorAFT which specifies whether the insertion shall be BEFORE or AFTER some other block which is already on the specified RING; (4) PLACE = a pointer to the block already on the RING. INSERT assumes that INSERTEE is not currently attached to the RING, and neither makes assumptions about nor has any effect on any other rings to which INSERTEE may be attached.

HEAD, TAIL, EMPTY -- HEAD, TAIL, and EMPTY are all functions and perform identical roles in all three handler packages, but have some differences in the way that they are called. First, as to function, HEAD returns a pointer to the head block of a ring, TAIL returns a pointer to the tail block of a ring, and EMPTY returns a Boolean TRUE if a ring is empty/ FALSE if not empty. The problem is how to specify the ring. In order to create a uniform, minimal calling sequence for these functions in the TTF version, we concocted a seventh, pseudo-sensor named NAUGHT (usually renamed GLOBAL), so that calling these functions with a real sensor causes them to refer to the sensor-oriented rings, while calling them with the pseudo-sensor GLOBAL causes them to refer to the global_ priority ring. As originally designed, the TCF blocks also had a sensor-oriented ranking and so this form of calling sequence was used there as well. In the course of development, this sensor-oriented ranking was abandoned, but the calling sequence was left unchanged to facilitate a future reintroduction of such a ranking if a need for it were perceived. In the PTL version, both rankings were global from the start, and so the calling sequence for PRIOTHLIST.HEAD and PRIOTHLIST. TAIL specified which of the two global rankings was intended. For function PRIOTHLIST.EMPTY, since both rankings are global, if either ring is empty, then they both are. So, for this last function, nothing is specified in the calling sequence.

CLEAR --The CLEAR procedure is intended for use by routines outside of the TRM proper, such as the TEST_BED and various interim, ad hoc drivers used during TRM debugging. Its intended purpose is to re-initialize the file blocks in preparation for a warm restart; it does this by DELETEing all file blocks currently on active duty.

RE-PRIORITIZE --This procedure INSERTs the file block pointed to by the input argument in its proper place on the global_priority ring. Priority ranking is descending in the forward ring direction.

RE-ARRANGE --This procedure exists only in package THREATFILE. It INSERTs the file block pointed by the input argument in its proper place in its sensor's azimuth-order ring. The sensor is picked up from the SIPDATA.SENSOR_ID field of the block. Azimuth ordering is ascending in the forward ring direction.

5.2.1.5. The CREC_HANDLER Package. This package contains the pool of available correlation records (COR_RECs) which are attached by the pointers COR_FIRST and COR_LAST to individual TCF records (TCF_ RECs) to indicate which TTF records (TTF_RECs) are collocated. The manner in which this attachment is made is as a doubly-linked chain, as discussed in an earlier

section. It also contains the following procedures and functions for handling these records:

OBTAIN --No difference whatever except for the type of pointer returned between this OBTAIN and procedures THREATFILE.OBTAIN and PRIOTHLIST.OBTAIN.

RELEASE --No difference whatever except for the object being RELEASEd between this RELEASE and procedure THREATFILE.RELEASE and PRIOTHLIST.RELEASE.

FREE_ONE --This function is the equivalent of the first form of the DETATCH procedures discussed above; it severs the chain attachments and reconnects the chain links remaining (if any), but leaves the block just free floating. As noted in an earlier section, the nature of the doubly-linked chain structure forces it to recognize five cases: chain already empty, chain is a singleton, block is being freed from the head, from the tail, from the middle of a non-trivial chain. In the first two of these cases, the function returns a Boolean FALSE, and in the last three a TRUE.

FREE_ALL --This procedure is the functional equivalent of the DELETE procedures discussed above, except that it performs its DELETE action on all blocks attached to the chain starters (COR_FIRST and COR_LAST) of a given TCF block. FREE_ALL uses FREE_ONE and RELEASE in that order inside a loop until FREE_ONE returns a FALSE value.

INSERT --This procedure exists in two overlayed versions. The first overlay is the functional equivalent of the INSERT procedures discussed above, i.e., it permits one to INSERT the INSERTEE BEFORE and AFTER a block (PLACE) already on the chain. There is no ring to be specified, but a pointer to the TCF block must be supplied in order to find the COR_FIRST and COR_LAST pointers. The logic must account for four cases: before an ordinary block, after an ordinary block, before the head block, and after the tail block. The second overlay always INSERTs the INSERTEE as the head block on the chain of a specified TCF block. It handles the case of an initially empty chain directly and uses the first overlay of INSERT to insert before the head block of a non-empty chain.

5.2.1.6. Ada and Dynamic Data Files -- Lessons Learned. The chief lesson learned in the course of Phase II with respect to the use of Ada for the design of dynamic data files and their handlers is the value of Ada features that were not available in the version of the compiler available to us. Foremost amongst these features was Ada generics, the ability for writing procedures independent of the types of their input/output arguments and to

"instantiate" a different version for each such type -- in essence, something like a high-order-language version of assembly language macros. Thus, for example, almost all of the handler procedures/functions could have been written in one generically typed version and instantiated four times. The lack of generics also encouraged a touch of non-uniform design that wouldn't have occurred otherwise. Thus, the need for CORRELFILE. OBTAIN and RELEASE to deal with the embedded COR-RECs would undoubtedly have been handled in a different manner than it actually was. The availability of generics and the compulsion toward uniformity that they engender would also undoubtedly have inspired a more uniform and less confusing handling of the calling sequences of the HEAD, TAIL and EMPTY functions across the three principal file types and their varying sensor-oriented and global ordering requirements.

It would have been possible to devise a Threat Resolution Module based on a monolithic dynamic data base, i.e., a data base with only one block type, containing the appropriate fields to represent all the states and relationships exhibited by the three-part data base described above. One can imagine using of an ordering ring to represent the aged-in subset of this monolithic data base. The reason that this was not done (back in Phase I) was to promote the decoupling of the component processes of the TRM and to reduce the size of critical regions in the data base with a view toward an eventual realtime implementation of the TRM, that is, an implementation of the TRM not as a set of subroutines (the present implementation) but as a set of loosely coupled concurrent tasks each operating on its own section of the dynamic data base. Although we have maintained the divided aspect of a realtime data base design, we cannot be sure that all the features we have designed into it will be consistent with good realtime design. A major point of worry is the extensive system of pointers linking the three sections of the dynamic data base; do these defeat decoupling? The answer is a guarded "yes," because the net effect of these pointers is to promote crossover: the ability of TRACK, say, to make inquiries into parts of the data base outside of the TTF which is its main concern. One can rest assured that the adaptation of the present data base to a realtime environment will ococupy a considerable part of our attention in Phase III.

5.2.2. The Static Data Base. The static data base is a respository for the many static (unchanging) data items and tables required to implement the Threat Resolution Module (TRM)in that style of software design termed "table-driven," i.e., able to cover a wide range of logical behavior with a minimal amount of actual code in which the logical course is determined by the values of data items stored in tables. These data items/tables include such things as the values of error windows (tolerances) for doing matching tests, Boolean flags which specify abilities such as whether or not a particular sensor is able to measure a parameter such as elevation and sets whose elements dictate which sensors can be correlated with which.

The static data base is contained in a single package named STATIC_DATABASE (file name: st-data.text) which is organized according to the particular TRM process (or subprocess within a process) which the various data items/

tables are used in more than one process; such cases will be pointed out in the sections which follow. This organization reflects the process-oriented development followed during Phase II and reflects also the fact that Phase II constituted a learning environment for preparing for Phase III.

Section (1) will discuss the form, content, and purpose of the static data as currently organized. Section (2) will discuss lessons learned during Phase II and extrapolate these lessons to some speculations on the organization of static data in Phase III.

5.2.2.1. Static Data: Form, Content and Purpose. The principal processes of the TRM are TRACK, CORRELATE, AGE_IN, AGE_OUT, and DECIDE_REACTION; the last process listed does not presently contribute to the static data. Within the TRACK process, the static data is classified in accordance with its use in two of TRACK's principal subprocesses: MATCH and ASSESS_LETHALITY. This organization will be followed in the paragraphs that follow.

Many of the data tables to be discussed are sensor-indexed, i.e., the set of index values is the following ordered set of enumeration literals named SENSOR_INDEX which is defined in and exported from package GEN_TYPES (file name: gen_types.text):

```
SENSOR_INDEX:  (LASER, NIS, OPTICAL, PMD, MM_WAVE, NBC)
```

With the exception of a special case in package AGE_IN_PAK, none of these indices is referred to outside of STATIC_DATABASE. The sensors are, instead, referred to generically via a variable usually named SENSOR; this implies a lack of specialized logic based on characteristics of a sensor that are not recorded in the static data base tables, and thus provides some indication that the goal of making the TRM be table-driven has been achieved.

When the data items/tables contain numeric values, these values are, in most cases, represented as floating point numbers. This choice of representation was governed by two considerations:

(i) The compiler we were using did not support fixedpoint real types, the probable ultimate design choice, and

(ii) Even if fixed-point real types had been available, the choice of floating point freed us to concentrate on the design of algorithms without having to worry about numeric accuracy issues. This convenience has been purchased at the cost of a somewhat degraded performance in that the floating point arithmetic operations are carried out by software subroutines and not by hardware instructions. In view of Phase II's goals, this seemed a worthwhile tradeoff.

In the paragraphs which follow, we shall usually avoid giving the specific numeric values provided in the tables, because in many instances these values are particularly prone to change as the performance of the TRM is "tuned-up."

a. Static Data for the TRACK Process. As noted, data provided for the TRACK process is classified under two of TRACK's principal subprocesses, MATCH and ASSESS_LETHALITY. Data supplied for MATCH are as follows:

CAN-MEASURE - This is a doubly-dimensioned array of Boolean flags which specify whether (true) or not (false) a particular sensor from the SENSOR_INDEX set can measure one of the three location parameters, AZIM (azimuth); ELEV (elevation); and RAYNJ (range -- this particular spelling chosen because "range" is a reserved word in the Ada language). Originally designed to serve the needs of MATCH, CAN_MEASURE is now also used in the following TRACK subprocesses: ANALYZE_MOTION and UPDATE_TTF; future plans call for its use in TRACK subprocess ASSESS_LETHALITY. CAN_MEASURE is also used by the CORRELATE process to govern the gathering of history SIGHTINGs.

Example: if CAN_MEASURE (SENSOR, RAYNJ) then....

AZIMUTH_TOLERANCE, ELEVATION_TOLERANCE, RANGE_TOLERANCE - Each of these is a sensor-indexed vector of floating point values used to determine whether a particular location parameter which is measurable by a particular sensor (see CAN_MEASURE above) in the input data matches similar data already recorded in a threat_tracking_file (TTF).

AZIMUTH_WEIGHT; ELEVATION_WEIGHT; RANGE_ WEIGHT - Each of these is a sensor-indexed vector of floating point values used to compute a match score according to the formula: Match_score: = Sigma_over_the_measurable_parameters of parameter_weight times parameter_deviation_squared; where parameter_deviation is 0 if the absolute difference of the two parameter values is within the parameter_ tolerance and equal to the absolute difference if this is within twice the parameter_tolerance; any difference exceeding twice the tolerance triggers a mismatch; all measurable parameters within tolerance yields a 0 score which triggers an immediate match (MATCH_WITHOUT_CHANGE).

ACCEPTABLE_SCORE - Assuming no immediate match has occurred, a match score may or may not have been accumulated by the time all candidates have been examined. If no score has been

accumulated, then the input is said to represent a GENUINE_NEWGUY. If a score has been accumulated, then it is the minimum of the scores generated. If this minimum score is less than or equal to ACCEPTABLE_ SCORE for the sensor in question, then the match is said to represent a MATCH_WITH_CHANGE, otherwise it is a POSSIBLE_NEWGUY which has to be subjected to motion analysis before its status can be finally declared.

This entire match algorithm is an experimental, heuristic projection toward what such an algorithm must eventually be. The interaction of the tolerances, weights and acceptable_scores is a subject for a future operational analysis study beyond the scope of the present effort.

Static data which support the ASSESS_LETHALITY subprocess of the TRACK process are as follows:

BASE_LETHALITY - This table is the only table in the static data base that is indexed by EMITTER_INDEX. This index is based on a presumed subtype of a sensor suppled by the sensor in its input to the TRM. The base-lethality is a floating point value which represents the intrinsic lethality of the sensor/emitter combination without respect to other important lethality-determining factors such as range, azimuth, and elevation.

AZIMUTH_LIMITS, AZIMUTH_MODIFIER; ELEVATION_LIMITS, ELEVATION_MODIFIER; RANGE_LIMITS, RANGE MODIFIER - These three pairs of tables are used to bring the location of a detected object to bear on the estimation of its lethality. Given a value for one of the three parameters, this value is matched against the PARAMETER_ LIMITS TABLE by subroutine INDEX_LOOKUP (package ELEM_FUNC) to determine an index L such that limit (L) < value < = limit (L + 1). This index is then used to look up a value from the PARAMETER_MODIFIER table (which has one fewer elements than the PARAMETER_LIMITS table). Once this has been done for all parameters, the assessed lethality is computed as the product of the base_lethality and the three looked-up modifiers. As presently designed, this modification is done irrespective of whether or not the sensor of the object whose lethality is being assessed CAN_ MEASURE all three parameters. This is a candidate for redesign early in Phase III. Each of these tables is indexed by a positive integer in the range 1..Locally-declared constant (a different constant for each table pair).

b. Static Data for the CORRELATE Process. Static data for the CORRELATE process consists of a single array indexed by SENSOR_INDEX; the elements of this array are somewhat complex records of type ABLE_SET. The name of the array is CAN_BE_CORRELATED; it was named for the intended functional purpose of the first two fields in the record. As development of the CORRELATE process unfolded, it was found convenient to insert newly perceived correlation_support data items into the record as additional fields. The constituent fields of the ABLE_SET record are as follows:

SENSET - This is a set as defined in package sets SETS_PACK. SENSOR _SETS (a package within a package), i.e., a vector of Boolean flags in one-to-one correspondence with the indices SENSOR_INDEX that indicate whether (true) or not (false) the sensor named by the index is a member of the set. The set SENSET belongs to a record corresponding to a particular (the "present") sensor, and is used to declare which other sensors can be correlated with the present sensor.

EMISET - This is a set as defined in package SETS_PACK.EMITTER _SETS (a package within a package), i.e., a vector of Boolean flags in one-to-one correspondence with the indices EMITTER_ INDEX that indicate whether (true) or not (false) the emitter named by the index is a member of the set. The set EMISET belongs to a record corresponding to a particular sensor, and is used to declare which of the emitter subtypes of that sensor are correlatable.

IS_PLATFORM - This is a Boolean flag used to indicate whether (true) or not (false) detection by the sensor whose index locates this record implies a platform, i.e., a mobile system emitting one or more sensor-detectable forms of energy. At present,the only record for which IS_PLATFORM is true is that corresponding to NIS.

PLAT_SPEED - This is a don't-care field if IS_PLATFORM is false, otherwise it represents the maximum speed of the platform in meters per second. This is used by TRACK_AIDS.ANALYZE_MOTION to determine whether the apparent separation of two objects could be due to the motion of either one of them. While not presently accomplished, future enhancements of the correlation algorithm include the integration of such motion analysis.

IS_ILLUMTOR - This is a Boolean flag used to indicate whether (true) or not (false) the sensor whose index locates this record is receiving energy originating from the

object. This is used by the TEST_WARNINGS subprocess of AGE_IN to post an ILLUMINATED warning for a hovering object.

AZTOL, RGTOL - These are floating point values that specify azimuth and range tolerances, respectively, for correlated objects. AZTOL is used by the CORRELATE processes to set up an azimuth search window in the azimuth ordered ring corresponding to a sensor which can be correlated with the sensor of the input data. It is also used by the MOTION_HISTORY subprocess of the AGE_IN process to discriminate between real motion and the random jitter of multiple observations. RGTOL is presently used only in this latter context. It will have a role in future enhancements of the correlation algorithm. Both AZTOL and RGTOL are set during the elaboration of package STATIC_DATABASE by internally defined functions named INITIALIZE_MAX_AZIMUTH_TOLERANCE and INITIALIZE_MAX_RANGE_TOLERANCE. Each of these selects the maximum of its respective parameter over the sensor and its SENSET correlation mates of the AZIMUTH_/RANGE_TOLERANCE factors used in MATCH.

MAX_SCORE - This is a floating point value whose intended purpose is to provide a maximum correlation score for comparing rival correlation candidates. MAX_SCORE's role would be analogous to that of ACCEPTABLE_SCORE in the MATCH subprocess of TRACK (see above). This has not been fully deveoped.

5.2.2.2. Static Data for the AGE_IN Process. Several categories of data used by AGE_IN but classified under CORRELATE were described above. The data categories which follow are used exclusively by AGE_IN; these include:

AGE_IN_COUNT - This is a sensor-indexed vector of integers each of which gives the number of times a particular object detected by that sensor must be seen before that object will be permitted to Age_In.

AGE_IN LETHALITY - This is a sensor-indexed vector of floating point values each of which gives the minimum lethality estimate which will enable Age_In. The Age_in_Count and the Age_In_Lethality are used together in a whichever-occurs-first manner.

SCOUT_HELI_WARNING_RANGE, ATTACK_HELI_WARNING_RANGE - These two floating point values give the minimum distance in meters that can be exhibited by a closing helicopter before the TEST_WARNINGS subprocess of AGE_IN will

post a SCOUT_NEARBY or HIND_ CLOSING warning. The posting of such a warning overrides any consideration of Age_In_Count or lethality. The distinction between a scout and an attack helicopter is made on the basis of the emitter field of the input from the NIS sensor.

5.2.2.3. Static Data for the AGE_OUT Module. The AGE_ OUT Module (AOM) is not a process of the TRM; it is an independent software module called from the same external software level that invokes the TRM itself. The TRM and the AOM share the data structures and the handlers of the dynamic data base which are described in another section of this document. The function of the AOM is to remove old or spent files from the dynamic data base. To do this, it inspects the time-ofarrival ordering of the prioritized_threat_ list (PTL) in oldest to youngest order looking for PTL records whose age (time elapsed since the object described was last seen) is beyond a certain limit (see below). In a future extension, the AOM will also look for PTL records that have been marked by the Reaction_Management Module as having had the prescribed reaction completed. The static data that supports these actions are as follows:

AGE_OUT_TIME - This is a sensor-indexed vector of times (a separate Ada type in this implementation) each element of which gives the maximum age that an object should attain before being eliminated.

MIN_AGE_OUT_TIME - This is a single time item representing the minimum value in the AGE_OUT_TIME vector. It is set up during elaboration of package STATIC_DATABASE by an internally defined procedure, INITIALIZE_MIN_AGE_OUT_ TIME. It is used to trigger an early exit from the time-oriented search loop: since the loop runs from oldest to youngest, once a PTL record is reached whose age is less than MIN_etc. no other PTL records are going to Age_Out.

5.2.2.4. Lessons Learned - Ada and Static Data Base Design. As indicated earlier, the design of the static data base was proccess-oriented: as each process was designed, the data items/tables required to support that process were designed and inserted into a catchall package for such data. We had largely achieved a table-driven design for the TRM. What we did not achieve in the way of design goals was a design that enables rapid reconfiguration of the software in terms of being able to eliminate and/or add sensors and/or reaction devices. There are three factors that account for this:

i. The process oriented approach described above:

ii. The assumption that this goal was not appropriate to Phase II;

iii. Allowing ourselves to be tempted into using attractive, but dangerous Ada features;

We shall concentrate on this latter point. The Ada feature in mind here is the ability to declare enumeration types and objects. These types/objects are quite useful in software like the TRM for such purposes as giving readable names to status indicators: Example:

```
type MOTION_STATUS is (INDETERMINATE, AWAY, CLOSING, HOVERING, NIL);
```

But when it came to naming the sensors, we allowed ourselves to fall into the same temptation; we defined an enumeration type called SENSOR_TYPES and its subtype SENSOR_INDEX, and therein lies the inability of the present data base design to achieve the design goal of rapid reconfigurability: it fixes the number and names of the sensors in the code itself instead of externalizing these matters in the data.

An approach to the design of the static data base aimed at achieving rapid reconfigurability, both for sensor/reactors and for tactical/error-recovery reasons, should be based on the following measures:

i. The design must be sensor-oriented not processoriented, i.e., instead of having a large number of function-oriented, sensor-indexed vectors as we presently have, we need to have a number of sensor-indexed super-records whose fields encompass the sorts of functions delineated in Section (1):

```
type SENSOR_RECORD is
record
     SENSOR_NAME          - string, maximum length TBD
     MEASURES_PARAMETER   - parameter-indexed array
                            of Boolean flags
     PARAMETER_TOL        - parameter-indexed array of
                            numeric tolerances
     PARAMETER_WEIGHT     - parameter-indexed array of
                            numeric weights
     etc.
end record;
```

ii. The sensors may still be specified in an enumeration type, but now they will be anonymous, i.e., SENSOR_1, SENSOR_2,... with the text that identifies the actual sensor embedded as data in the SENSOR_RECORD, as shown above. Some maximum number of sensors will be provided for, with the actual number presently in the system being recorded as a global variable < = this maximum.

Under these design measures, reconfiguration, whether at a dynamic level or at the level of recompilation of the code, would replace whole SENSOR_RECORDs and adjust the variable that specifies how many are present.

5.2.3. The Threat Resolution Module.

5.2.3.1. Introduction and Overview. The Phase II version of the Threat Resolution Module (TRM) is structured as a procedure named THREAT_RESOLUTION (package TRM_PACK, file name: trm_pack.text) which is largely a logical skeleton for invoking the principal processes TRACK, CORRELATE, AGE_IN, and DECIDE_REACTION. These processes are themselves, in turn, also structured as procedures which may invoke lower level procedures (subprocesses). Associated with the TRM is the AGE_OUT Module (AOM) which is not a process of the TRM but an independent software module called from the same external software level that invokes the TRM itself. The overall block diagram of the TRM is shown in Figure 5-15.

The TRM and the AOM are file-oriented modules. The major function of the TRM is to create, update, and maintain a set of dynamic data files which collectively represent a perception of the external battlefield situation. (The term "file", as used here, means a structured collection of data records stored in the memory of the DMS computer, i.e., an internal file). The three principal types of dynamic data files are the Threat_Track Files (TTFs) created and maintained by the TRACK Process, the Threat_Correlation Files (TCFs) created and maintained by the CORRELATE Process, and the Prioritized_Threat List (PTL) created and maintained by the AGE_IN Process. The function of the AOM is to remove old or spent files from the dynamic data base. Thus, the relationship between the TRM and the AOM is that they share the data structures and the procedures/functions which handle the various data file types listed. The internal details of the dynamic data base and its handlers are provided in Paragraph a. of this Section 5.2. (Page 22).

The actual functional behavior of the TRM and the AOM is in large measure determined by the settings of various flags, numeric values, and other types of data stored as the tables/items of the static data base (package STATIC_DATABASE, file name: st_data.text). This design approach results in what is termed "table-driven" software. The internal details of the static data base are provided in Section 5.2.2. (Page 50).

The overall logic of the TRM presented in the skeletal framework of TRM_PACK will be discussed in Paragraph 2. Paragraphs 3 through 7 will provide discussions of the principal processes and their subprocesses (for the purposes of this document, AGE_OUT will be classified as a TRM process):

PARA NO.	PROCESS NAME	PACKAGE NAME	FILE NAME
3	TRACK	TRACK_PACK	trak_pack.text
4	CORRELATE	CORR_PACK	corr_pack.text
5	AGE_IN	AGE_IN PAK	agein_pack.text
6	DECIDE-REACTION	REAC_PACK	reac_pack.text
7	AGE_OUT	AGO_PACK	ago_pack.text

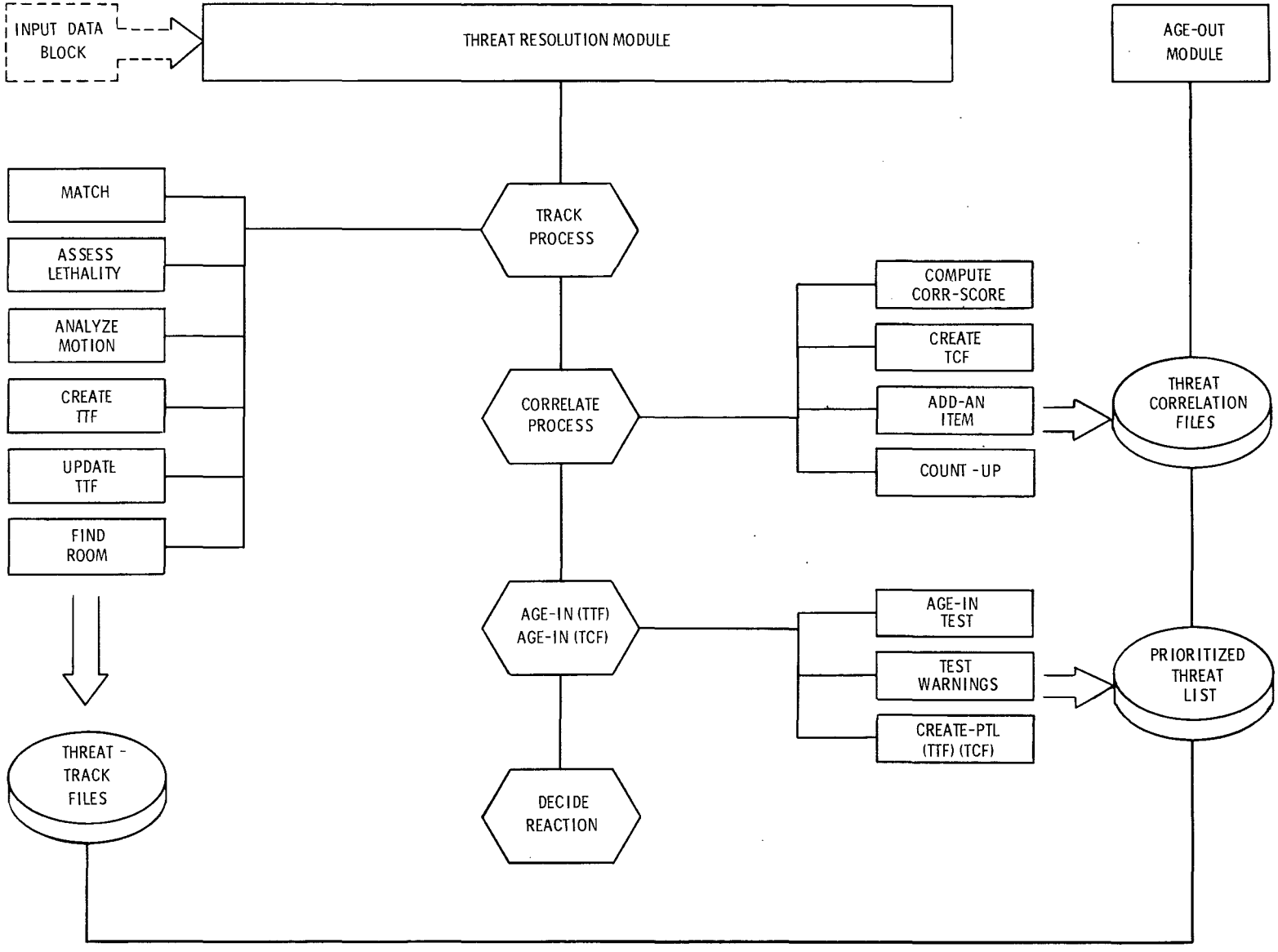


Figure 5-15. Threat Resolution Module (TRM)

5.2.3.2. TRM Overall Logic. Procedure THREAT_RESOLUTION constitutes the executable code of the TRM; it is the sole export of package TRM_PACK, and it accepts as arguments two pointers and the current time:

Input Arguments:	INPUT_PTR
	OUTPUT_PTR
	CLOCK_TIME

The INPUT_PTR points to the first of a chain of one or more INPUT_DATA_BLOCKS (IDBs):

INPUT_PTR-->	I D B	-->	I D B	. . .	I D B
	#1		#2	. . .	LAST

Each IDB consists of a pointer and a SENSOR_INPUT_PACKET (SIP); both the IDB and the SIP are defined in and exported from SIP_ - PACK. The pointer is used to point to the next IDB if the IDB in which it appears is not the last in the chain or, contrariwise, to mark the end of the chain (with value = null). The SIP portion of the IDB contains information received from a sensor; all IDBs on a given input chain pertain to the same sensor. This input design was adopted in the expectation that some sensors could hand off multiple observations. The information contained in a SIP is as follows:

SIP_RECORD:	SENSOR_ID	- Name/index of sensor
	EMITTER	- Platform/threat subclass
	RANGE_WORD	- Distance in meters
	AZIMUTH	- In degrees
	ELEVATION	- In degrees
	TIME_SENSED	- See note below

Each of these fields is present for every sensor without regard for whether or not a given sensor can measure a particular parameter. At the level of Phase II, we have made the convenient assumption that the units in which a particular parameter is expressed are the same for all sensors; the three location parameters are all represented as floating point numbers. See further remarks on this subject in Paragraph b (Page 54).

The OUTPUT_PTR points to a list of pointers to Prioritized_ Threat List entries which have become newly aged_in as a result of this call on the TRM.

The CLOCK_TIME is the time at which the TRM is being called. Each of the principal dynamic data base file types has a field for recording the latest time seen; any of these file entries which is created or updated as a result of this call on the TRM will have this time inserted into said field. The TIME_SENSED field of the input SIPs measure the time at which the BUS_INPUT_PROCESS (outside the TRM) received the SIP and is provided for purposes of throughput-time measurements in a future version providing access to a realtime clock.

THREAT_RESOLUTION serves only as a skeleton of the required processing logic, invoking processes (procedures) in other packages in pursuit of its ends. The packages containing these subsidiary procedures are provided in square brackets in the structured-English description of the logic of THREAT_RESOLUTION which follows:

```
**********************************************

                THREAT_RESOLUTION

            STRUCTURED_ENGLISH DESCRIPTION

**********************************************

Set the CURRENT_INPUT_PTR equal to the INPUT_PTR argument

Repeat the following loop until the last IDB has been processed,
     i.e., until CURRENT_INPUT_PTR = End_of_Chain marker (null)

     Set THIS_SIP equal to the SIP portion of the IDB pointed to
     by the CURRENT_INPUT_PTR

     Invoke the TRACK process [TRACK_PACK] providing it with the
     following inputs:  THIS_SIP and CLOCK_TIME, getting back the
     following outputs:  TRACK_FILE and TRACKING_FLAG

     --REMARK:  If TRACK successfully matches THIS_SIP to the
                SIPDATA field of an existing Threat_Track File
                (TTF), then TRACK_FILE will point to same and
                TRACKING_FLAG will have the value UPDATED;

          OR: if TRACK has established a new TTF, then TRACK_
              FILE will point to same and TRACKING_FLAG will
              have the value CREATED;

          OR: if THIS_SIP described a low priority threat and
              there was no room left to establish a new TTF,
              then TRACK_ FILE will have no meaning (will be null)
              and TRACKING_FLAG will have the value DISCARDED.

     If the TRACKING_FLAG has the value DISCARDED, then skip to
     the paragraph just before the end of the loop ("Replace...").

         If the TRACK_FILE was not previously correlated, then
         invoke the CORRELATE process [CORR_PACK] providing the
              following inputs:  TRACK_FILE, getting back the
              following outputs: CORR_ FILE
         else set CORRL_FILE to null so that the subsequent logic
              will behave as though correlation did not occur.
```

```
        If correlation occurred:
        then invoke the AGE_IN process [AGE_IN_PAK] providing the
             following inputs:  CORRL_FILE, getting back the
             following outputs:  AGED_IN_FILE
        else invoke the AGE_IN process [AGE_IN_PAK] providing the
             following inputs:  TRACK_FILE, getting back the
             following outputs:  AGED_IN_FILE.

      --REMARK:  There are two different versions (Ada
                 overlays) of the procedure AGE_IN; they are
                 distinguished by their input arguments

                 In either case, if age_in does not occur, the
                 output AGED_IN_FILE is a null pointer.

        If Age_In occurred:
        then invoke the DECIDE_REACTION process [REAC_PACK]
             providing the
             following inputs:  AGED_IN_FILE, getting back the
             following outputs: - TBD -

      (This point marks the end of the if TRACKING_FLAG statement)

      (Replace CURRENT_INPUT_PTR with the IBD pointer pointed to
      by CURRENT_INPUT_PTR, if this is the End_of_Chain marker,
      this will be discovered at top of loop.

      End of the loop and of the THREAT_RESOLUTION procedure.

    ************************************************************
```

Note the non-realtime nature of this design; there is no notion of concurrency of processes. Each SIP is pushed through all the stages which its parameter values and the current state of the dynamic data base will allow, before the next SIP can be considered. This means that a current SIP of low priority can keep a SIP of immense importance waiting.

5.2.3.3. The Track Process. This package defines and exports the procedure named TRACK and the enumeration type, TRACK_RESULT whose three values, DISCARDED, UPDATED and CREATED where shown being used in Paragraph b. TRACK is also somewhat skeletal, accomplishing its major actions through procedures contained in package TRACK_AIDS (file name trak-aids.text). Both TRACK and its auxiliaries will be detailed in this paragraph, starting with TRACK itself to illustrate the functions of the auxiliaries in an operational context.

As noted in Paragraph b, THREAT_RESOLUTION sends TRACK a SIP_ RECORD received as input and the CLOCK_TIME at the time of call, and receives in return a pointer to a Threat_Track File (TTF) entry and a TRACKING_FLAG taking one of the three TRACK_RESULT values listed above. If TRACKING_FLAG

has the value DISCARDED, then TRACK_FILE will be a null (meaningless) pointer; if the value is UPDATED, then TRACK_FILE will point to a matching TTF which has just been updated with the information contained in the input SIP_RECORD (name: THIS_SIP); if the value is CREATED, then TRACK_FILE will point to a TTF entry which has just been created with the input data of THIS_SIP. Achieving an understanding of the logic that connects the inputs to these possible outcomes is our next task. To that end, we present a structured English description of the TRACK process with interspersed remarks. All procedures invoked in this description are found in package TRACK_AIDS unless otherwise noted in square brackets.

****

TRACK_PROCESS

STRUCTURED_ENGLISH DESCRIPTION

****

Invoke the MATCH procedure providing it with
the following inputs: THIS_SIP, getting back
the following outputs: TRACK_FILE and MATCH_FLAG

--REMARKS: This is essentially where all of TRACK's work is done; everything that follows is in reaction to the combinations of values that the output values take.

Output MATCH_FLAG is of type MATCH_RESULT which is defined in and exported from TRACK_AIDS. The values it can take will become apparent in the logic which follows.

The first value of MATCH_FLAG tested for is DISCARD. The reponse to this outcome is to set TRACKING_FLAG to DISCARDED and return.

--REMARKS: There is presently no logic in MATCH which causes MATCH_FLAG: = DISCARD, but this value was introduced against a future situation in which MATCH would also perform various validity checks on the input and use this outcome to flag unacceptable input.

At this point, some sort of a match has been achieved, so the next thing done is to invoke procedure ASSESS_LETHALITY which accepts THIS_SIP as input and provides a floating point value, LETHALITY, as output.

-- REMARK: The logic which follows has been simplified in the interest of enhanced understanding by introducing a quantity of fictitious "goto" statements. The flow of control is actually handled by putting a case statement inside an infinite loop and getting out of this loop in three different ways: (a) exit statements, (b) return statements, and (c) changing the value of the case expression, so that the next time around the loop, the case encountered ends in a return. The head of the loop is visited a maximum of two times.

The other outcomes for MATCH_FLAG and the responses to them are given in the following numbered clauses. The reasons for MATCH_FLAG taking any particular value will be given when we discuss the MATCH procedure.

o MATCH_FLAG = GENUINE_NEWGUY

Goto section below labelled NEWGUY_LOGIC

o MATCH_FLAG = POSSIBLE_NEWGUY

Invoke the ANALYZE_MOTION procedure, providing it the following inputs: THIS_SIP, TRACK_FILE, getting back the following outputs: MOTION_FLAG (type MOTION_ RESULT defined in/exported from TRACK_AIDS).

a. MOTION_FLAG = NEWGUY

(Match discrepancy can't be explained by motion).

Goto section below labelled NEWGUY_LOGIC

b. MOTION_FLAG = OLDGUY

(Match discrepancy is explained by motion).

Goto Case 3, immediately following.

o MATCH_FLAG = MATCH_WITH_CHANGE

Invoke the UPDATE_TTF procedure, providing it with the following inputs: THIS_SIP, LETHALITY, CLOCK_ TIME, TRACK_FILE, and Update-degree = FULL,

getting back

the following outputs: TRACK_FILE

Set the TRACKING_FLAG to UPDATED and return.

o MATCH_FLAG - MATCH_WITHOUT_CHANGE

Invoke the UPDATE_TTF procedure, providing it with
the following inputs: THIS_SIP, LETHALITY,
CLOCK_ TIME, TRACK_FILE, and
update_degree = PARTIAL,

getting back

the following inputs: TRACK_FILE

Set the TRACKING_FLAG TO UPDATED and return.

-- REMARK: The update_degree parameter is of type UP_
DEGREE defined in and exported from TRACK_AIDS
taking the two values FULL and PARTIAL.

NEWGUY_LOGIC

Invoke procedures OBTAIN [THREATFILE] which has no input
arguments but returns the pointer TRACK_FILE.

If there is no room for a new TTF (TRACK_FILE = null)
then invoke procedure FIND_ROOM, providing it with
the following inputs: LETHALITY, getting back
the following outputs: TRACK_FILE

If TRACK_FILE still = null,
then no room can be found, therefore,
set the TRACKING_FLAG to DISCARDED and return
else Clear out the contents of the room which has been
found at TRACK_FILE

--REMARK: For further insight into the matter of
running out of file space see the discussion
in Section a,(Page 26) and the documentation of
procedure FIND_ROOM, below.

Arriving at this point, we have a pointer to an empty TTF
entry, either from OBTAIN or from FIND_ROOM:

Invoke procedure CREATE_TTF, providing it with
the following inputs: THIS_SIP, LETHALITY, CLOCK_TIME and
TRACK_FILE, getting back
the following outputs: none

Set the TRACKING_FLAG to CREATED and return

End of TRACK process.

****

a. The MATCH Subprocess. The MATCH subprocess (procedure) accepts as input a SIP_RECORD named THIS_SIP and attempts to match the data in the SIP (sensor input packet) to the same data items in the SIPDATA field of all TTF entries pertaining to the same sensor, i.e., the search is conducted over a same_sensor subset of the TTF, and that sensor is the one given in the SENSOR_ID field of the SIP_RECORD and in the SIPDATA of the TTF entries. MATCH returns a MATCH_FLAG and in the SIPDATA of the TTF entries. MATCH returns a MATCH_FLAG to TRACK. This takes one of the values DISCARD, GENUINE_NEWGUY, POSSIBLE_NEWGUY, MATCH_ WITH_CHANGE and MATCH_WITHOUT_ CHANGE. Except for the first two of these values, MATCH also returns a valid (non-null) pointer (TRACK_FILE) to the best match available, where the meaning of "best" will be explained in the next paragraph.

MATCH_FLAG = DISCARD is intended (in a future version) to denote rejection of invalid input data. A value of GENUINE_ NEWGUY indicates no match found. The other values indicate the degree of match attained. At the high end, MATCH_WITHOUT_CHANGE indicates that all parameters matched within the permitted windows (tolerances), and is used by TRACK to trigger a PARTIAL update of TRACK_FILE. Next comes MATCH_WITH_CHANGE which indicates that the former degree of match could not be attained, but what was possible was a match in which no parameter discrepancy exceeded twice the prescribed tolerance AND of all such matches TRACK_FILE had the smallest discrepancy-based score AND this score did not exceed a prescribed ACCEPTABLE_SCORE. MATCH_WITH_CHANGE is used by TRACK to trigger a FULL update of TRACK_FILE. Finally, a rating of POSSIBLE_NEWGUY indicates that all of the conditions for MATCH_WITH_CHANGE were met except the last one. Here the possibility exists that the apparent discrepancy between the parameters of THIS_SIP and those of TRACK_FILE.SIPDATA are due to the motion of one, the other or both of the objects between observations. Thus, TRACK uses such an indication to call on ANALYZE_MOTION to rule on the acceptability of this motion hypothesis.

The simplified summary of the preceeding two paragraphs is made more precise in the structured_English description which follows:

```
**************************************************

                MATCH_SUBPROCESS

          STRUCTURED_ENGLISH DESCRIPTION

**************************************************

PRELIMINARIES:  Set TRACK_FILE to null so that if a MOTION_ FLAG =
                DISCARD or GENUINE_NEWGUY exit is taken, no TTF
will be indicated.  Set SENSOR to the SENSOR_ID field of THIS_SIP.

Test whether the azimuth_ordered ring pertaining to SENSOR is
empty, i.e., test the Boolean value returned by function
THREATFILE.EMPTY -- if this is true, then set MATCH_FLAG =
GENUINE_NEWGUY and return.
```

LOOP SETUP: Set ponters S_PTR and S_LAST to the head and tail entries, respectively, of the azimuth-ordered ring pertaining to SENSOR. Set the MATCH_PTR to null and the MATCH_SCORE to +Infinity.

Repeat the following loop until one of its exit conditions is met.

Set the SCORE TO 0

If SENSOR cannot measure azimuth, proceed to the ELEVATION_MATCH_TEST

Compute DELTA_1 as the absolute difference between THIS_SIP.AZIMUTH and S_PTR.SIPDATA.AZIMUTH.

If DELTA_1 falls within the AZIMUTH_TOLERANCE for this SENSOR, then proceed to ELEVATION_MATCH_TEST (nothing added to score)

orif DELTA_1 is within twice said tolerance
then augment SCORE by the AZIMUTH_WEIGHT for this SENSOR times the square of DELTA_1; proceed to ELEVATION_MATCH_TEST.

else: The failure to match could be due to azimuth wraparound from 360 degrees to zero, compute DELTA_2 as the absolute difference between DELTA_1 and 360.

Repeat the previous two tests with DELTA_2 replacing DELTA_1, if both of these tests fail, proceed to NEXT_FILE.

ELEVATION_MATCH_TEST: If SENSOR cannot measure elevation, proceed to RANGE_MATCH_TEST.

Compute DELTA_1 as the absolute difference between THIS_SIP.ELEVATION and S_PTR.SIPDATA.ELEVATION.

If DELTA_1 falls within the ELEVATION_TOLERANCE for this SENSOR, then proceed to RANGE_MATCH_TEST (nothing added to SCORE).

orif DELTA_1 is within twice said tolerance,
then augment SCORE by the ELEVATION_WEIGHT for this SENSOR times the square of DELTA_1; proceed to RANGE_MATCH_TEST.

else Proceed to the NEXT_FILE.

RANGE_MATCH_TEST: If sensor cannot measure range, proceed to EMITTER_MATCH_TEST.

```
        Compute DELTA_1 as the absolute difference between THIS_SIP.
        RANGE_WORD and S_PTR.SIPDATA.RANGE_WORD.

        If DELTA_1 falls within the RANGE_TOLERANCE for this SENSOR,
        then proceed to EMITTER_MATCH_TEST (nothing added to SCORE).

        orif DELTA_1 is within twice said tolerance,
        then augment SCORE by the RANGE_WEIGHT for this SENSOR times
            the square of DELTA_1; proceed to EMITTER_MATCH_TEST.

        else proceed to the NEXT_FILE.

        EMITTER_MATCH_TEST:     We will tolerate a mismatch of the
                                emitter type if and only if the score
        attained so far is within reasonable limits:

        If SCORE does not exceed the ACCEPTABLE_SCORE for this SENSOR
        then proceed to INSPECT_SCORE,
        else proceed to NEXT_FILE

        INSPECT_SCORE:

        If SCORE is still equal 0,
        then All parameters matched within tolerance; so
            Set MATCH_FLAG to MATCH_WITHOUT_CHANGE,
            Set TRACK_FILE to S_PTR, and
            Return. ----------------------------Loop Exit 1--->

        Otherwise, if SCORE is less than MATCH_SCORE

        then Replace MATCH_SCORE by SCORE and
            Set MATCH_PTR to S_PTR.

        NEXT FILE:

        If S_PTR is equal to S_LAST,
        then Exit from the loop ------------------- Loop Exit 2--->
        else make S_PTR point to the next TTF in the azimuth-ordered
            ring for this SENSOR.

End of the Loop.

POST_LOOP_LOGIC:

If the MATCH_PTR is still null,
then all excursions of the loop abort at "proceed to NEXT_FILE"
     without a SCORE being obtained; so Set
     MATCH_FLAG to GENUINE_NEWGUY and
     Return.
```

```
orif MATCH_SCORE does not exceed the ACCEPTABLE_SCORE for this
     SENSOR,
then Set MATCH_FLAG to MATCH_WITH_CHANGE,
     Set TRACK_FILE to MATCH_PTR, and
     Return.

else, Set MATCH_FLAG to POSSIBLE_NEWGUY,
      Set TRACK_FILE to MATCH_PTR, and
      Return.

      End of the MATCH Subprocess.

          ******************************************
```

The MATCH subprocess was designed before it was decided to create an azimuth_within_sensor ordering on the TTFs. A redesign of MATCH to take advantage of this ordering is not a trivial task, and so is left in the category of desirable, future enhancements. An example of a file search that does take advantage of this ordering may be found in the CORRELATE process.

b. The ASSESS LETHALITY Subprocess. ASSESS_LETHALITY is called with THIS_SIP (the input Sensor Input Packet) as its argument and it returns as its output argument LETHALITY, a floating point value which measures the estimated lethality of the threat object represented by THIS_SIP. ASSESS_LETHALITY is called from TRACK when it is clear that some sort of a match has been declared by MATCH (MATCH_ FLAG not equal DISCARD). The estimated lethality generated here follows the input SIP through the course of its further processing by the TRM. A brief list of the functions in which LETHALITY is an active participant is as follows:

- It is stored in the TRPIO field of a TTF when CREATE_TTF is called by TRACK to create a new TTF entry and updated when TRACK calls UPDATE_TTF;

- It is stored in the CPRIO field of a Threat_Correlation File (TCF) entry and represents the maximum over all the TRPIO fields of the TTFs associated with the TCF;

- It is stored in the PPRIO field of a Prioritized_Threat List (PTL) entry and represents the maximum TPRIO over all the TTFs constituting an aged_in object:

- It is used to keep all three of the above file types on priority_ ordered rings;

- It is used in the FIND_ROOM Subprocess (below) to determine if there are files with lesser priority which can be released to make room to create a new TTF.

Estimated lethality is based on four factors: (a) The emitter type of the object, (b) its range, (c) its azimuth, and (d) its elevation. Factor (a) (THIS_SIP.EMITTER) is used to index the STATIC_DATABASE array, BASE_LETHALITY. The other factors are used in a search function, ELEM_FUNC. INDEX_LOOKUP, to find an index for obtaining the three parameter_MODIFIERs (parameter = RANGE, AZIMUTH, ELEVATION). The final estimate is the product of the base_lethality and the three modifiers.

Each location parameter is represented by two tables, a parameter_LIMITS table and a parameter_MODIFIER table. The lookup process is the same for all three parameters and will be illustrated for AZIMUTH only. The AZIMUTH_LIMITS and MODIFIER tables are as follows:

INDEX	LIMITS ENTRY
1	-1.00
2	45.00
3	135.00
4	225.00
5	315.00
6	361.00

INDEX	MODIFIER ENTRY
1	64.00
2	16.00
3	4.00
4	16.00
5	64.00

The INDEX-LOOKUP function searches for the position i of value = THIS_SIP.AZIMUTH in the Limits table such that

$$\text{Limit}(i) < \text{Value} <= \text{Limit}(i + 1)$$

Thus, an object at 90 degrees azimuth would yield an index of 2, and this index into the MODIFIER table would fetch a modifier of 16.00.

As presently constituted, ASSESS_LETHALITY does not take cognizance of whether or not sensor THIS_SIP.SENSOR_ID measure all of the parameters considered above. This, too, is a planned future enhancement.

c. The ANALYZE MOTION Subprocess. ANALYZE_MOTION is called by TRACK when MATCH returns with MATCH_FLAG set to POSSIBLE_ NEWGUY. ANALYZE_MOTION rules or whether the parameter discrepancy represented by the POSSIBLE_NEWGUY status is due to motion by one, the other, or both of the objects represented by THIS_SIP and TRACK_FILE (the two input arguments to ANALYZE_MOTION). If the ruling is in favor of motion, the MOTION_FLAG output takes the value OLDGUY, otherwise it takes the value NEWGUY.

This analysis of motion is based on the assumption that the VIDS vehicle is stationary, and it ignores the truly significant effect of the speed of sound, the form of energy detected by one of the most important range_measuring sensors. It also ignores elevation as a factor and pays no attention to any time elapsed or motion occurring since the most recent of the two observations.

Given those assumptions, the algorithm consists of computing the inputted travel of the object using the law of cosines, then computing the apparent speed of the object by dividing this inputed travel by the time elapsed between the two observations and finally deciding whether this is a reasonable speed for the type of object in question. If the speed is reasonable, then we infer that the apparent discrepancy in location represents motion of the object. If the speed is not reasonable (i.e., is too large), the second observation is taken to be a NEWGUY.

In order to do this limited analysis, the two candidate objects, THIS_SIP and TRACK_FILE, must both be detected by sensors that can measure range. At least one of them must be detected by a sensor which implies a platform (i.e., something possessing a maximum speed). It is assumed that a platform is a range_measurer, but not the reverse. If the TRACK_FILE does not produce the sought-for property directly, it may do so indirectly by being correlated to a platform or a range_measurer.

ANALYZE_MOTION is designed to handle two major cases:

1. THIS_SIP and TRACK_FILE were detected by the same sensor. This case arises when A_M is called from TRACK, and reduces the qualification tests of the previous paragraph to: the common sensor must imply a platform.

2. THIS_SIP and TRACK_FILE were detected by different sensors. This case is intended to serve the needs of the CORRELATE process, but this integration has not yet taken place. This is also an area for future enhancement.

We shall simplify the discussion which follows by confining ourselves to the logic which handles the first of these cases (much of which is shared with the second case). This logic is found in the structured_English description which follows:

****

ANALYZE MOTION SUBPROCESS

STRUCTURED_ENGLISH DESCRIPTION

****

Assuming the qualifications for the called_from TRACK case have been met:

Take the SPEED factor from the platform corresponding to the common sensor. Set up the primary computational factors required as follows:

```
TF_RANGE    from TRACK_FILE.SIPDATA.RANGE_WORD

SIP_RANGE   from THIS_SIP.RANGE_WORD

DEL_T       from THIS_SIP.TIME_SENSED - TRACK_FILE.TTOA

DEL_AZ      from THIS_SIP.AZIMUTN - TRACK_FILE.SIPDATA.AZIMUTH
```

Then use the Law of Cosines to compute the imputed travel:

```
TRAVEL is SQRT (SIP_RANGE**2 + TF_RANGE**2
                -2.0 * SIP_RANGE * TF_RANGE * COS (DEL_AZ))
```

Finally, compare the imputed speed to the maximum platform speed:

```
If TRAVEL / DEL_T does not exceed SPEED,
then set MOTION_FLAG to OLDGUY,
else set MOTION_FLAG to NEWGUY

Return

End of ANALYZE_MOTION Subprocess.
```

****

d. The CREATE TTF Subprocess. CREATE_TTF is called by TRACK when MATCH has declared a GENUINE_NEWGUY or ANALYZE_MOTION has ruled out motion with a MOTION_FLAG = NEWGUY. All arguments to CREATE_TTF are input arguments; these are: THIS_SIP, TRACK_FILE, LETHALITY, and CLOCK_TIME.

The logic of CREATE_TTF is arrestingly simple and has insufficient structure to merit an extended structured_English description. The logic of CREATE_TTF is as follows:

```
Set TRACK_FILE.SIPDATA     from      THIS_SIP
Set TRACK_FILE.TTOA        from      CLOCK_TIME
Set TRACK_FILE.TPRIO       from      LETHALITY
Set TRACK_FILE.AICNT       to        1   (Age_In count)

Invoke procedure THREATFILE.REPRIORITIZE to insert TRACK_FILE
on the priority (lethality) ordered ring.

Invoke procedure THREATFILE.REARR_ANGLE to insert TRACK_FILE
on the azimuth_within_sensor ordered ring.
```

e. The UPDATE TTF Subprocess. UPDATE_TTF is called by TRACK to perform two degrees of update on TTF entries. A FULL update is requested when MATCH returns with MATCH_FLAG set to MATCH_WITH_CHANGE and a PARTIAL update is requested when MATCH_FLAG is set to MATCH_WITHOUT_CHANGE or when MATCH_FLAG is set to POSSIBLE_NEWGUY and the subsequent call on ANALYZE_MOTION

produces an OLDGUY result. The chief differences between the two degrees of update is that a FULL update updates the position parameters in the TTF's SIPDATA field and makes adjustments in the file ordering rings for changes in azimuth and priority (lethality).

The arguments presented to UPDATE_TTF are:

THIS_SIP	- SIP record from input
LETHALITY	- Computed by ASSESS_LETHALITY
CLOCK_TIME	- From input
TRACK_FILE	- The TTF being updated
UP_OPTION	- FULL or PARTIAL

The following structured_English description provides a more detailed look at the update logic:

```
**************************************************

                UPDATE_TTF SUBPROCESS

            STRUCTURED_ENGLISH DESCRIPTION

**************************************************

If UP_OPTION is PARTIAL then goto PARTIAL_UPDATE.

Set N to the floating point representation of TRACK_FILE's
age_in count field (AICNT).

--REMARK:  The following logic uses a procedure named
           NEW_AVERAGE (package ELEM_FUNC) which updates an
           average as follows:

           NEW_AVERAGE( Avg, N, Val)

              ==> Avg: = (N*Avg + Val) / (N + 1)

If the sensor THIS_SIP.SENSOR_ID can measure range,
then NEW_AVERAGE ( Avg, N, Val ) where:

     Avg is TRACK_FILE.SIPDATA.RANGE_WORDK
     Val is THIS_SIP.RANGE_WORD
If the sensor can measure elevation,
then NEW_AVERAGE( Avg, N, Val ) where

     Avg is TRACK_FILE.SIPDATA.ELEVATION
     Val is THIS_SIP.ELEVATION

If the sensor can measure azimuth
then Set Avg  from TRACK_FILE.SIPDATA.AZIMUTH,
     Set Val from THIS_SIP.AZIMUTH.
```

```
      Update azimuth recognizing possibility of wraparound:

      If Val > = 315 degrees and Avg < = 45 degrees,
      then NEW_ AVERAGE( Avg, N, Val-360°)
           if Avg is now negative, add 360° to it.

      orif Val < = 45 degrees and Avg  > = 315 degrees,
      then NEW_ AVERAGE( Avg, N, Val+360),
           if Avg is now > 360, reduce it by 360°.

      else  NEW_AVERAGE( Avg, N, Val )

      Set TRACK_FILE.SIPDATA.AZIMUTH from Avg,
      DETATCH TRACK_FILE from its azimuth ring,
      Reattach TRACK_FILE in azimuth order (REARR_ANGLE)

      If the current TRACK_FILE priority (TRACK_FILE.TPRIO) is less
         than THIS_SIP's LETHALITY
      then Reset TRACK_FILE.TPRIO from LETHALITY
           Reset TRACK_FILE.SIPDATA.EMITTER from THIS_SIP.EMITTER
           DETATCH TRACK_FILE from its priority ring
           Reattach TRACK_FILE in priority order (RE_PRIORITIZE)

PARTIAL_UPDATE:

--REMARK:  The following logic uses an internally defined
           procedure UPDATE_PTL(PTL) which does the
           following things:

           - Sets PTL's latest_time_seen field PTL.PTOA from
             TRACK_FILE.TTOA
           - DETATCHes PTL from its time_of_arrival ring
           - INSERTs PTL at the head of said ring

           - If the PTL's priority field (PTL.PPRIO) is less
             than TRACK_FILE.TPRIO
             then Reset PTL.PPRIO from TRACK_FILE.TPRIO
                  DETATCH PTL from its priority ring
                  Reattach PTL in priority order (RE_PRIORITIZE).

Increment TRACK_FILE's age_in_count field (AICNT),
Set TRACK_FILE's latest_time_seen field from CLOCK_TIME

If TRACK_FILE is already correlated (TRACK_FILE.TTCFP not null),
then Set COR_FIL from TRACK_FILE.TTCFP.

     If TRACK_FILE.TPRIO > COR_FIL.CPRIO,
     then Reset COR_FIL.CPRIO from TRACK_FILE.TPRIO
          DETATCH COR_FIL from its priority ring
          Reattach COR_FIL in priority order (RE_PRIORITIZE)
```

```
        Set COR_FIL's latest_time_seen field from TRACK_FILE.TTOA

        If COR_FIL is aged_in (COR_FIL.CPTLP not null),
        then UPDATE_ PTL (COR_FIL.CPTLP) -- see remark above.

        If the sensor THIS_SIP.SENSOR_ID can measure range
        then Invoke TRM_TYPES. SAVE_HISTORY to record a new SIGHTING
             in COR_ FIL

    orif TRACK_FILE is aged_in (TRACK_FILE.TPTLP not null),
    then UPDATE_PTL (TRACK_FILE.TPTLP).


    End of UPDATE_TTF Subprocess.

      ************************************************************
```

f. The FIND ROOM Subprocess. FIND_ROOM is called by TRACK when a call on THREATFILE.OBTAIN yields a null TTF_PTR indicating that there is no room left to establish a new TTF. The basic thrust of FIND_ROOM is to search for an existing TTF that has a priority lower than input argument LETHALITY. If such a file is found, it is DETATCHED or DELETEd so that its room can be claimed by TRACK for the new higher priority TTF. The file space thus freed is pointed at by output argument TRACK_FILE. If the quest is unsuccessful, FIND_ROOM returns with TRACK_FILE set to null.

The logic of FIND_ROOM is made more precise by the following structured_ English description:

```
      ************************************************************

                          FIND_ROOM SUBPROCESS

                      STRUCTURED_ENGLISH DESCRIPTION

      ************************************************************

    --REMARK:   The search for a disposable file is carried out in two
                passes, with the second pass not used if the first
                succeeds.  In the first pass, we search for an
                uncorrelated TTF whose priority (FYLE.TPRIO) is less
                than LETHALITY.  If the  TTF is also aged_in, then FYLE
                must also pass the P_TEST.  If P_FILE is set from
                FYLE.TPTLP, then P_TEST (P_FILE) is true if:

                  a.   P_FILE.STATUS is COMPLETED, or

                  b.   P_FILE.STATUS is not ACTIVE and
                       P_FILE.PPRIO is less than LETHALITY.
```

```
FIRST_PASS -- TTF_SEARCH:

Set TRACK_FILE to null in anticipation of a vain search

Set up pointers for a low-to-high scan of the TTF priority ring

   Set FYLE        to point at the priority ring tail block
   Set LAST_FILE   to point at the priority ring head block

While FYLE.TPRIO > = LETHALITY, repeat this loop

     If FYLE is not correlated (FYLE.TTCFP is null),
     then If FYLE is not aged_in (FYLE.TPTLP is null),
          then DETATCH FYLE from all of its ring attachments
          Set TRACK_FILE from FYLE
          Return
     orif:  P_TEST (FYLE.TPTLP) is true,
     then DELETE ( FYLE.TPTLP )
          DETATCH FYLE from all its ring attachments
          Set TRACK_FILE from FYLE
          Return

     exit from this loop if FYLE = LAST_FYLE
     otherwise, set FYLE to point to the TTF with the next
                    highest priority

End of TTF_SEARCH loop

--REMARK:   Having arrived at this point, the TTF_SEARCH has
           failed; therefore, we undertake the second pass in
           which we first test to see if there are any Threat_
           Correlation_Files (TCFs), and if so we do a TCF_
           SEARCH which differs from the previous one only in
           the complexity of getting rid of the files found to
           be disposable.

SECOND_PASS -- TCF_SEARCH:

If there are no TCFs present, simply return

Set up pointers for a low_to_high scan of the TCF priority ring:

     Set TCFL      to point at the priority ring tail block
     Set LAST_TCFL to point at the priority ring head block

While TCFL.TPRIO > = LETHALITY, repeat this loop

     If TCFL is aged_in (TCFL.CPTLP not null)
     then Set PTLIT from TCFL.CPTLP
          If P_TEST ( PTLIT ) is true
          then DELETE (PTLIT)
               Goto MAKE_ROOM
          else goto MAKE_ROOM
```

```
        exit from this loop if TCFL = LAST_TCFL
        otherwise, set TCFL to point to the TCF with the next
                   highest priority

    End of TCF_SEARCH loop

    --REMARK:  If we arrive at this point, both passes have failed

    Return

    ------------------------------------------------------------

    MAKE_ROOM:  First we must DELETE all the TTFs that are correlated
                with the TCF (TCFL) we have just found.  Then we can
                DELETE TCFL itself.  Finally, we can again call
                THREATFILE.OBTAIN which will not fail this time.

    Set CITEM from TCFL.COR_FIRST; it now points to the first
        correlated item record

    Repeat the following loop until the exit condition is met

        DELETE the TTF pointed to by the COR_ITEM field of the record
        (CITEM.COR_ITEM)

        exit if this is the last correlated item record
        (CITEM.NEXT is null)

        otherwise, reset CITEM from CITEM.NEXT

    End of correlated item loop

    DELETE the TCFL

    OBTAIN a TTF and return it as TRACK_FILE

    Return

    End of FIND_ROOM Subprocess

    ************************************************************
```

5.2.3.4. The CORRELATE Process. Correlation is the process of discovering that two different sensors have detected an object that is at the same physical location (within reasonable error windows related to the accuracy with which the sensors involved measure the parameters that specify physical location). Once such a positive finding has been established, the TTFs involved are said to be correlated or members of a correlated object.

The CORRELATE process is defined in and is the sole export from package CORR_PACK. All of its auxiliary subprocesses are defined in the body of CORR_PACK. CORRELATE is called with input argument TRACK_FILE (pointer to the TTF we are seeking to correlate), and it returns with output argument CORR_FILE (pointer to the TCF with which TRACK_FILE correlates or null to indicate no correlation).

CORRELATE is called by THREAT_RESOLUTION after TRACK has been called and has returned with a TRACKING_FLAG which is not set to DISCARDED. One further condition must be set before CORRELATE may be called: the TTF, TRACK_FILE, must not be already correlated, i.e., the pointer TRACK_FILE. TTCFP must be null. This will, of course, be true if TRACK_FILE has just been created, but there may be cases when an OLDGUY is not correlated. These conditions may be regarded as dynamic conditions; other conditions of a more static nature are imposed on correlatability in the course of the CORRELATE process itself. These conditions as well as other factors affecting the course of the CORRELATE process are contained in a STATIC_DATABASE array named CAN_BE_CORRELATED which is SENSOR_INDEXed. Specific descriptions of this static data are given in Section B.1(b) above.

We will begin our description of the CORRELATE process with a structured_English amplification of the process's logic, giving operational definition to the static data items and the subprocesses. After that we will provide descriptions of these subprocesses: COMPUTE_CORR_SCORE, CREATE_TCF, ADD_AN_ITEM, and COUNT_UP.

****

CORRELATION

STRUCTURED_ENGLISH DESCRIPTION

****

PRELIMINARY DEFINITIONS:

THIS	:	The SIP_RECORD, TRACK_FILE.SIPDATA
AZTH	:	THIS.AZIMUTH
SENSOR	:	THIS.SENSOR_ID
MITTER	:	THIS.EMITTER
PERMISSIONS	:	CAN_BE_CORRELATED (SENSOR)
IF_IS_PLAT	:	PERMISSIONS.IS_PLATFORM
WRAP_AROUND	:	A Boolean flag initialized to False to indicate that the aximuth search zone does not straddle the 0 to 360 degrees wrap-point; may be changed later.

CORRELATE LOGIC:

We start by setting CORRL_FILE to null in anticipation of a failure to correlate.

Then we extract a set (in the full sense of set theory) named S_MATES from the PERMISSIONS record (field SENSET). S_MATES names the other sensor types which can correlate with SENSOR, and we simply abort (return from) the process at this point if S_MATES is empty. See SETS_PACK discussion in Paragraph 8 below.

Only some of SENSOR's emitter types may be correlatable. Therefore, we obtain set PERMISSIONS.EMISET, and return from the process at this point if MITTER is not a member of this set.

Next, we define the lower and upper limits of the azimuth search zone as LIM_INF: = AZTH - WINDOW while LIM_SUP: = AZTH + WINDOW, where WINDOW is the AZTOL field of PERMISSIONS.

Then, we determine if WRAP_AROUND obtains: WRAP_AROUND is True if either LIM_INF < 0 degrees or LIM_SUP > 360 degrees. In the first case, we augment LIM_INF by 360 degrees and in the second case, we decrease LIM_SUP by 360 degrees.

As the last preliminary before entering the MAIN_LOOP, we set CORRELATION_SCORE to +Infinity and WINNER to null.

--REMARKS: CORRELATION_SCORE is intended to be the analog of MATCH_SCORE and WINNER the analog of MATCH_PTR in the MATCH subprocess of TRACK. The analogous logic is not fully developed in the current version of CORRELATE.

The logic of the MAIN_LOOP as currently coded involves six levels of if statements and inner loops. In order to simplify and clarify the description of this logic, we shall:

a) introduce some fictitious Goto's and labels, and
b) describe the action of certain inner loops without structuring them formally as loops.

MAIN_LOOP: For S in SENSOR_INDEX loop:

Let S_IS_PLAT be a Boolean flag which indicates whether or not sensor S implies a platform. This is taken from the IS_PLATFORM field of CAN_BE_CORRELATED(S).

We now have to impose some constraints on the eligibility of Sensor S to correlate with SENSOR:

FIRST : S must be a member of S_MATES (see above)

SECOND : The azimuth_ordered ring pertaining to sensor S must not be empty: not THREATFILE.EMPTY(S)

THIRD : SENSOR and S cannot both imply a platform: not (TF_IS_PLAT and S_IS_PLAT)

If any of these strictures is violated
then Goto NEXT_SENSOR (at end of MAIN_LOOP)

--REMARK: The search for WINNER takes place in two parts. The forward_search does the entire search if WRAP_AROUND is False, otherwise it accomplishes the search from 0 degrees up to LIM_SUP. The backward_search takes place only if WRAP_AROUND is True; here the search retrogresses from 360 degrees down to LIM_INF.

The complexity introduced into the logic by the need to allow for azimuth wrap_around could possibly be reduced by introducing two azimuth orderings, one the same as the present to be used when no wrap_around has occurred, and the second running from -180 degrees up to +180 degrees to be used when wrap_around has occurred. This is an item for future consideration.

FORWARD_SEARCH: -- Fictitious label:

Set T_FILE to point at the head block of the azimuth_ordered ring for sensor S (block with smallest azimuth).

It is possible that the forward search is not necessary. This could arise if the smallest azimuth is already larger than LIM_SUP; if this is so, goto BACKWARD_SEARCH.

Otherwise, set T_LAST to point at the tail block of the azimuth ordered ring for sensor S (block with largest azimuth).

If WRAP_AROUND is False, enter a small loop whose purpose is to move the pointer T_FILE forward in the ring until T_FILE points to the first block whose azimuth is > = LIM_INF. It is possible to run out of blocks (T_FILE = T_LAST and azimuth of T_LAST block is < LIM_INF). If this occurs, goto NEXT_SENSOR.

We are now ready to inspect the forward part or the entirety of the azimuth search zone:

While T_FILE.SIPDATA.AZIMUTH < = LIM_SUP loop

Inside this inner_loop we call the subprocess COMPUTE_CORR_ SCORE if a final set of qualification tests can be passed. The first of these tests rules out T_FILE if its emitter type is not correlatable. The second test eliminates T_FILE if T_FILE is correlated to a platform, even though T_FILE's own sensor does not imply a platform.

Exit the MAIN_LOOP if CORRELATION_SCORE is now = 0.

Exit this loop if T_FILE = T_LAST

Move T_FILE forward to the next lowest azimuth block in the ring.

End of loop.

BACKWARD_SEARCH: -- Fictitious label

If WRAP_AROUND is false goto NEXT_SENSOR

Otherwise, set T_FILE to point at the tail block of the azimuth_ ordered ring for sensor S (block with largest azimuth).

It is possible that the backward search is not necessary. This could arise if the largest azimuth is already less than LIM_INF; if this is so, goto NEXT_SENSOR.

Otherwise, set T_LAST to point at the head block of the azimuth_ ordered ring for sensor S (block with smallest azimuth).

We are now ready to inspect the backward part of the azimuth search zone:

While T_FILE.SIPDATA.AZIMUTH > = LIM_INF loop

Inside this inner_loop we call the subprocess COMPUTE_CORR_ SCORE if a final set of qualification tests can be passed. The first of these tests rules out T_FILE if its emitter type is not correlatable. The second test eliminates T_FILE if T_FILE is correlated to a platform, even though T_FILE's own sensor does not imply a platform.

Exit the MAIN_LOOP if CORRELATION_SCORE is now = 0

Exit this loop if T_FILE = T_LAST.

```
        Moving T_FILE backward to the next highest azimuth block in
        the ring.

      End of loop.

    NEXT_SENSOR:  a null statement to allow real goto's to reach
                                    the end of the loop

End of MAIN_LOOP.

At this point, there exist two possibilities for failure to cor-
relate.  The first is that WINNER was never reset by COMPUTE_CORR_
SCORE from its initial null value.  The second is that there is
a finite CORRELATION_SCORE, but it is greater than the accept-
able score in PERMISSIONS.MAX_SCORE.  If either condition obtains,
return from this point.

Next, we set up LETHALITY as the

  Maximum of WINNER.TPRIO and TRACK_FILE.TPRIO
             if WINNER is not CORRELATED

  Maximum of WINNER.TTCFP.CPRIO and TRACK_FILE.TPRIO
             if WINNER is correlated

LETHALITY will be passed as a global variable to CREATE_TTF and
ADD_AN_ITEM since these subprocesses are defined within the scope
of CORRELATE.

Finally,

If WINNER is not correlated (WINNER.TTCFP is null)
then Invoke CREATE_TTF
else Invoke ADD_AN_ITEM

CORRELATES's output argument, CORRL_FILE, will be returned directly
from one or the other of these internally defined subprocesses.

End of CORRELATE Process

*****************************************************************
```

a. The COMPUTE CORR_SCORE Subprocess. The COMPUTE_CORR_SCORE subprocess is called from those inner_loops of the MAIN_LOOP that pertain to examining the TTF blocks in the azimuth search zone. Defined within the scope of CORRELATE, it receives only the input argument CANDIDATE which is actualized as the current value of the pointer T_FILE. It updates the non-argument variables CORRELATION_SCORE and WINNER.

In a fully matured design of a correlation process, this subprocess would contain the logic that does the actual matching within windows of the other (non-azimuth) location parameters, making allowance for whether or not both sensors (S and SENSOR) can measure a given parameter. The net result of these comparisons would be a penalty_ score (larger is worse) which would be compared to CORRELATION_SCORE and would replace CORRELATION_SCORE if it were less than the latter. At the same time, WINNER would be set from T_ FILE. Thus, at the end of the MAIN_LOOP, WINNER would either be null or point to the TTF meeting all correlation conditions and having the best match to TRACK_FILE. The logic of CORRELATE proper has been designed to integrate properly with such a subprocess design. The TRACK subprocess, ANALYZE_MOTION, has been designed to serve CORRELATE in the same functional role that it plays with respect to TRACK/MATCH, requiring only a small amount of additional logic in CORRELATE.

The present design of COMPUTE_CORR_SCORE is a simplified stand-in for the design sketched above; it merely sets the CORRELATION_SCORE to 0 and sets WINNER from CANDIDATE. The net result of these actions is: If correlation is possible, it will occur with WINNER equal to the first T_FILE that falls inside the azimuth search zone. Another consequence is to render meaningless the post-MAIN _LOOP test of CORRELATION_SCORE versus PERMISSIONS.MAX_ SCORE; it will never fail. This algorithm, despite its crudity, has been of value in debugging CORRELATE and its interface to the entire dynamic data base.

b. The CREATE_TCF Subprocess. CREATE_TCF is called at the end of CORRELATE when WINNER is a confirmed correlate of TRACK_ FILE and WINNER is not already correlated (WINNER.TTCFP is null). It obtains space for a new TCF block and initializes all fields therein. It also revises pointers in WINNER and TRACK_FILE. CREATE_TCF is defined within the scope of CORRELATE and receives and returns all it requires without a calling sequence. In particular, unless CORRELATE ends by calling ADD_AN_ITEM (see below), CREATE_TCF is responsible for returning a non-null CORRL_FILE pointer to THREAT_RESOLUTION via CORRELATE's calling sequence.

The logic of CREATE_TCF is explained in the following structured_English description:

****

CREATE_TCF

STRUCTURED_ENGLISH DESCRIPTION

****

Invoke CORRELFILE.OBTAIN to get a pointer CORRL_FILE to an unused TCF block. Section A explains why there is no danger of running out of TCF blocks. Section A also indicates that obtaining a TCF also causes two COR_RECs to be obtained via CREC_HANDLER.OBTAIN and attached to the chain starters COR_ FIRST and COR_LAST which are fields of the TCF block.

Record the pointers TRACK_FILE and WINNER on the chain just mentioned:

```
CORRL_FILE.COR_FIRST.COR_ITEM: = TRACK_FILE
CORRL_FILE.COR_LAST.COR_ITEM:  = WINNER
```

Call subprocess COUNT_UP to set the RANGER, GUIDERS, and history fields of the TCF. This requires two calls: first for WINNER because it is the older and then for TRACK_FILE because it is the younger of the two TTFs.

Set up the IS_PLAT field of CORRL_FILE as the logical (inclusive) or of the two Booleans CAN_BE_CORRELATED (X).IS_PLATFORM where X is a sensor index taking successively the values: the sensor pertaining to TRACK_FILE and the sensor pertaining to WINNER.

Set up the CPRIO field of CORRL_FILE from the value of LETHALITY computed in the post-MAIN_LOOP logic of CORRELATE. Then call CORRELFILE.RE_PRIORITIZE to install CORRL_FILE on the TCF priority ring. No prior DETATCH is required, as in ADD_AN_ITEM, because CORRL_FILE having just been obtained, has no ring attachments.

Set CORREL_FILE's latest_time_seen field, CTOA, from the TTOA field of TRACK_FILE. This latter time was set from CLOCK_TIME in CREATE_TTF or in UPDATE_TTF.

This section of logic which we are now entering has as its object, creation/revision of the dynamic data base superstructures discussed in Section A.

First, we make both TRACK_FILE and WINNER point at CORRL_FILE.

```
TRACK_FILE.TTCFP: = CORRL_FILE
WINNER.TTCFP:     = CORRL_FILE
```

Then, we handle the possibility that age_in may be present in one, the other, both or neither of TRACK_FILE and WINNER. If neither is aged_in, there is nothing left to do. Otherwise, the other three cases are handled using an internal_to_CREATE_TCF procedure named RECONNECT. The logic of RECONNECT will be given after its use is demonstrated.

```
If TRACK_FILE is aged_in (TRACK_FILE.TPTLP not null), but
   WINNER isn't,
then RECONNECT( TRACK_FILE )

orif  WINNER is aged_in, but TRACK_FILE isn't,
then RECONNECT( WINNER )
```

```
orif  both WINNER and TRACK_FILE are aged_in,
then PRIOTHLIST.DELETE (the younger)
RECONNECT (the older of the two).

-- REMARK:  This criterion for deciding which PTL block to delete
            and which to reconnect is an area for early redesign
            effort.  For one thing, it takes no cognizance of the
            status and warning fields they contain.

Formal end of CREATE_TCF logic.

-----------------------------------------------------------------

RECONNECT:

Let the formal argument be PTR, and let P_FILE be set from
PTR.TPTLP.

Make CORRL_FILE point at P_FILE:  CORRL_FILE.CPTLP: = P_FILE.

Sever P_FILE's connection to PTR: P_FILE.PTTFP: = null.

Make P_FILE point at CORRL_FILE:  P_FILE.PTCFP: = CORRL_FILE

Set up P_FILE's time and priority fields:

     Set P_FILE.PTOA from TRACK_FILE.TTOA
     Set P_FILE.PPRIO from LETHALITY (as above)

Finally, DETATCH P_FILE from its time_of_arrival ring connections,
reINSERT P_FILE as the head block on this same ring

     DETATCH P_FILE from its priority ring connections, call RE_
     PRIORITIZE to insert it in priority order thereon

End CREATE_TCF Subprocess

     ***************************************************
```

c. The ADD_AN_ITEM Subprocess. ADD_AN_ITEM is called at the end of CORRELATE when WINNER is a confirmed correlate of TRACK_FILE, but is already correlated (WINNER.TTCFP non-null). It obtains space for a new COR_REC in which to record TRACK_FILE as a new member of the correlated object, and attaches this record to the existing chain in the TCF (WINNER.TTCFP). It also revises various superstructure pointers in TRACK_FILE, and updates various TCF fields. ADD_AN_ITEM is defined within the scope of CORRELATE and receives and returns all it requires without a calling sequence. In particular, unless CORRELATE ends by calling CREATE_TCF (see above), ADD_AN_ITEM is responsible for returning a non-null CORRL_FILE pointer to THREAT_RESOLUTION via CORRELATE's calling sequence.

The logic of ADD_AN_ITEM is explained in the following structured_English description:

```
**********************************************************

                        ADD_AN_ITEM

                STRUCTURED_ENGLISH DESCRIPTION

**********************************************************

Set CORRL_FILE from WINNER.TTCFP

Call CREC_HANDLER.OBTAIN to get a pointer, CREC, to an unused COR_
REC.
Call CREC_HANDLER.INSERT to attach CREC to CORRL_FILE.

Record TRACK_FILE as a new member of this correlated object:
       CREC.COR_ITEM: = TRACK_FILE

Make TRACK_FILE point at CORRL_FILE:  TRACK_FILE.TTCFP: = CORRL_
     FILE.

At this point, ADD_AN_ITEM requires, but currently does not have
logic similar to that done in CREATE_TCF to handle age_in:
TRACK_FILE and CORRL_FILE -- one, the other, both, or neither
aged_in.  This may involve making RECONNECT external to CREATE_TCF
so that it is accessible to ADD_AN_ITEM.

Call subprocess COUNT_UP to set the RANGER, GUIDERS, and history
fields of the TCF.  This requires only one call:  for TRACK_FILE.

Set up the IS_PLAT field of CORRL_FILE as the logical (inclusive)
or of its current value and CAN_BE_CORRELATED (sensor of TRACK_
FILE). IS_PLATFORM.

Set up the CPRIO field of CORRL_FILE from the value of LETHALITY
computed in the post-MAIN_LOOP logic of CORRELATE.  Next, call
CORRELFILE.DETATCH to sever CORRL_FILE's priority ring connections,
after which call CORRELFILE.RE_PRIORITIZE to install CORRL_FILE on
this same ring.

Set CORRL_FILE's latest_time_seen field, CTOA, from the TTOA field
of TRACK_FILE.  This latter time was set from CLOCK_TIME in
CREATE_TTF or in UPDATE_TTF.

End of ADD_AN_ITEM Subprocess

**********************************************************************
```

d. The COUNT_UP Subprocess. COUNT_UP is a small piece of logic called from both CREATE_TCF and ADD_AN_ITEM. Its function is to update the RANGERS, GUIDERS, and history fields of CORRL_FILE with the information borne by the TTF of the input argument TTFIL. The RANGERS field is an integer which tells how many of the TTFs correlated into this TCF can measure range. The GUIDERS field is an integer which tells how many of these TTFs illuminate a target. The history is a group of fields which registers: (a) HISCOUNT = how many SIGHTINGs have been recorded, (b) YOUNGEST = index of most recent SIGHTING, and (c) SIGHTING = an array of HISTO_RECs containing fields: WHO = pointer to the TTF sighted, RNGE = range, AZIM = aximuth, and TYME = time. SIGHTING is a circular array in that once HISCOUNT reaches its maximum value (5), each new sighting overwrites the currently oldest sighting. The task of saving the history is done by procedure SAVE_HISTORY in package TRM_TYPES.

The logic of COUNT_UP is explained in the following structured_English description:

```
**************************************************

                    COUNT_UP

            STRUCTURED_ENGLISH DESCRIPTION

**************************************************

Set SENSOR from TTFIL.SIPDATA.SENSOR_ID -- TTFIL's sensor.

If SENSOR can measure range (CAN_MEASURE (SENSOR, RAYNJ) is true),
then Increment CORRL_FILE.RANGERS,
     Call SAVE_HISTORY (TTFIL, CORRL_FILE)

If SENSOR implies illuminator (IS_ILLUMTOR field of
                               CAN_BE_ CORRELATED (SENSOR) is true)
then increment CORRL_FILE.GUIDERS

End of COUNT_UP subprocess.

**************************************************
```

5.2.3.5. The AGE_IN Process. The AGE_IN process is actually a pair of processes, one for aging_in uncorrelated TTFs and another for aging_in correlated objects (TCFs). Both processes have the name AGE_IN; the Ada language is able to distinguish between these two "overlays" by analyzing the declared types of the calling sequence arguments. The first of these processes is invoked by THREAT_RESOLUTION after the CORRELATE process has been called to consider an uncorrelated TRACK_FILE and the TRACK_FILE re-Omains uncorrelated; here, the input argument is TRACK_FILE. The second of these processes is invoked by THREAT_RESOLUTION if the TRACK_FILE was already or has just become correlated; here, the input argument is CORRL_FILE, the TCF to which TRACK_FILE is correlated. Both processes return the

same output argument, AGED_IN_FILE, which is a pointer to the PTL representing the establishment of the input file's aged_in status or null if age_in did not occur. For the sake of clarity and brevity, we shall refer to the two processes as AGE_IN(TTF) and AGE_IN(TCF).

The two processes, AGE_IN(TTF) and AGE_IN(TCF), are not independent, since the former calls the latter. The two processes rely on a number of subprocesses: two overlayed versions of CREATE_PTL (CREATE_PTL(TTF) and CREATE_PTL(TCF)), function TEST_WARNINGS and function AGE_IN_TEST. The text which follows will present the logic of the two processes in the order AGE_IN(TTF) followed by AGE_IN(TCF); the logic of the subprocesses will follow these.

```
**************************************************

                    AGE_IN (TTF)

              STRUCTURED_ENGLISH DESCRIPTION

**************************************************


SUMMARY:    If the input TRACK_FILE is already correlated, then we
            attempt to age_in the correlated object to which it
            belongs.  Otherwise, we call AGE_IN_TEST to determine
            if TRACK_FILE  can be aged_in, and if so, we call
            CREATE_PTL(TTF) to create a new AGED_IN_FILE (result
            pointer).  If age_in is not possible, the result pointer
            is set null.

If TRACK_FILE already correlated (TRACK_FILE.TTCFP not null)
then Invoke AGE_IN(TCF), providing
     the following inputs:  TRACK_FILE.TTCFP, getting back
     the following outputs: AGED_IN_ FILE.

orif AGE_IN_TEST (TRACK FILE) is true
then Call PRIOIHLIST.OBTAIN to get AGED_IN_FILE as a pointer to
          an unused PTL block
     Call CREATE_PTL(TTF) with inputs TRACK_FILE, AGED_IN_FILE to
          set up its fields and establish its ring connections

else Set AGED_IN_FILE to null
Return

End AGE_IN (TTF) Process

**************************************************************
```

****

AGE_IN (TCF)

STRUCTURED_ENGLISH DESCRIPTION

****

Set AGED_IN_FILE to null in anticipation of a failure to age_in.

If the correlated object is already aged_in (CORRL_FILE.CPTLP not
null)
then Return (nothing left to do)

First, see if a special warning can be posted:

WARNING: = TEST_WARNINGS (CORRL_FILE) -- function invocation

If the value returned is not the NULL_WARNING
then Goto AGE_IT_IN

Otherwise, we allow the whole correlated object to age_in if any of its component TTFs is able to age_in. To do this, we must make an excursion of the chain of COR_RECs which point to CORRL_FILE's associated TTFs.

Let CITEM be the COR_PTR to the first link on the chain:
CITEM: = CORRL_FILE.COR_FIRST

Repeat the following loop until its exit condition is met

If AGE_IN_TEST (CITEM.COR_ITEM) is true,
then Goto AGE_IT_IN

Exit when the pointer to the next link (CITEM.NEXT) is null

Otherwise, set CITEM to CITEM.NEXT

End loop.

Arriving at this point implies that no age_in occurred, therefore

Return

```
AGE_IT_IN:

Call PRIOTHLIST.OBTAIN to get AGED_IN_FILE as a pointer to an
     unused PTL block

Call CREATE_PTL (TCF) with inputs CORRL_FILE, AGED_IN_FILE to
     set up its fields and establish its ring connections

Put the WARNING previously obtained into the PFLAG (warning flags)
     field of AGED_IN_FILE

Return

End of AGE_IN(TCF) Process

*********************************************************************


    ******************************************************

                          AGE_IN_TEST

                  STRUCTURED_ENGLISH DESCRIPTION

    ******************************************************

This function accepts input argument TRACK_FILE and returns a
Boolean flag = true if age_in can occur, false otherwise.

TRACK_FILE can be aged_in (return value true) if

     _ It is not already aged_in
       [ TRACK_FILE.TPTLP is null ] and

     - Either:  It has been seen the required number of times
                [ TRACK_FILE.  ACINT > = AGED_IN_COUNT( SENSOR )]

          or:   Its lethality estimate is sufficiently large
                [ TRACK_FILE.TPRIO > = AGE_IN_LETHALITY ( SENSOR )]

where SENSOR is TRACK_FILE.SIPDATA.SENSOR_ID

  and AGE_IN_COUNT, AGE_IN_LETHALITY are SENSOR_INDEXed arrays
      in the STATIC_ DATABASE.

End AGE_IN_TEST subprocess.

   ******************************************************
```

****

TEST_WARNINGS

STRUCTURED_ENGLISH DESCRIPTION

****

SUMMARY: TEST_WARNINGS is a function whose input argument, CORRL_FILE, is pointer to a TCF representing a correlated object. It returns a warning of type REAC.TYPES.WARNINGS which may take one of the following values:

NULL_WARNING
ILLUMINATED
HIND_CLOSING
SCOUT_NEARBY
UNIDENTIFIED_OBJECT_CLOSING

The selection of which of these warnings to post is based on an analysis of CORRL_FILE's motion history contained in the TCF history fields HISCOUNT, YOUNGEST and SIGHTING. (These should be reviewed in Section B and under the COUNT_UP subprocess of CORRELATE in Paragraph C.4.d.) This analysis is done by an internally defined function MOTION_HISTORY which also accepts CORRL_FILE as its input argument and returns a value of locally-defined type MOTION_STATUS, taking one of the values:

INDETERMINATE
AWAY
CLOSING
HOVERING

MOTION_HISTORY is a long and somewhat laborious routine which is better described by summarization than elaboration of its logical structure. This will be done after the logic of TEST_WARNINGS proper.

These warnings are concerned with NIS-detectable and other objects which can be detected by range-measuring sensors. Therefore, we are interested in this correlated object if the sensors that detected it include at least one range-measurer. A further criterion is that the correlated object must have a full SIGHTING array (NHIST = 5 entries):

If CORRL_FILE.RANGERS = 0 or CORRL_FILE.HISCOUNT < NHIST,
then Goto NULL_RETURN (whence a NULL_WARNING will be returned)

```
Otherwise, we call MOTION_HISTORY(CORRL_FILE) and use its returned
value in a case statement:

Case     MOTION_HISTORY (CORRL_FILE) is:

     when HOVERING -- We are interested in knowing if the hovering
                      object is illuminated as:

         If CORRL_FILE.GUIDERS = 0
         then Goto NULL_RETURN
         else Return ILLUMINATED

     when CLOSING  -- We want to post different warnings depending
                      on whether NIS is involved and within NIS,
         yet other warnings depending on the subspecies that the
         NIS sensor is able to distinguish.  The first thing that
         must be done is to find out if NIS is involved:

     Set NIS_FOUND, a Boolean flag, to false

     Let CITEM be the COR_PTR to the first link of the chain of
     COR_RECs that record the TTFs belonging to this correlated
     object:  CITEM: = CORRL_FILE.COR_FIRST

     Repeat the following loop until one of its exit conditions
       is met:

       Set NIS_FOUND to true if
                        CITEM.COR_ITEM.SIPDATA.SENSOR_ID = NIS.

       Exit if NIS_FOUND is true

       Exit when the pointer to the next link (CITEM.NEXT) is null

       Otherwise, set CITEM to CITEM.NEXT

     End loop

     -- REMARK:  This determination could be done more simply in
                 a future version by adding a Boolean flag, say
                 IS_NIS, to the TCF_REC definition and setting
                 this to true in the COUNT_UP subprocess of
                 CORRELATE when a newly correlated TTF comes from
                 NIS.

     At this point NIS_FOUND has its proper value and can be tested:

     If NIS_FOUND is still false
     then Return UNIDENTIFIED_OBJECT_CLOSING
```

Otherwise, we are dealing with a closing NIS-detectee, and CITEM.COR_ITEM points to the TTF whose sensor is NIS. The rest of the analysis is concerned with the emitter subtype that the NIS sensor can distinguish. For the sake of this paradigmatic alogrithm we have assumed that these subtypes are

    FRIEND_HELI
    ATTACK_HELI (assumed to be a HIND)
    SCOUT_HELI

Set MITTER from CITEM.COR_ITEM.SIPDATA.EMITTER

If MITTER = FRIEND_HELI
then Goto NULL_RETURN

At this point, we have either an ATTACK or a SCOUT_HELI. We need to know if the object is close enough to warrant the posting of a warning. First, we dip into the history fields of the correlated object to get the range of the most recent SIGHTING:

DISTANCE := CORRL_FIL.SIGHTING(CORRL_FILE.YOUNGEST).RNGE

Then, we compare this distance to the relevant warning ranges:

If MITTER = SCOUT_HELI and
   DISTANCE <= SCOUT_HELI_WARNING_RANGE [*]
then Return SCOUT_NEARBY

orif MITTER = ATTACK_HELI and
     DISTANCE <= ATTACH_HELI_WARNING_RANGE [*]
     then Return HIND_CLOSING

else Goto NULL_RETURN

[*] These are items in the STATIC_DATABASE

When any other value of MOTION_STATUS is returned

Goto NULL_RETURN

End of case statement.

NULL_RETURN: Return NULL_WARNING

Formal end of TEST_WARNINGS logic

MOTION_HISTORY SUMMARY: The declaration of HOVERING is decided first, on the basis of a clustering analysis. If the difference between the maximum and the minimum range is less than a certain range tolerance, AND if the difference between the maximum and maximum azimuth (taking cognizance of wrap_ around) is less than a certain azimuth tolerance, THEN return a value of HOVERING.

Otherwise, convert the azimuth/range coordinates to Cartesian coordinates and attempt to fit a straight line to these points. If the fit is bad, return a value of INDETERMINATE. If the points are collinear, we compute DEL_RNGE as the difference between the most recent and the oldest range in the SIGHTING array. Then, assuming that TOL_RG is a positive range tolerance, we declare the outcome as follows:

```
     If DEL_RNGE < -TOL_RG, then Return CLOSING

     Orif DEL_RNGE > TOL_RG, then Return AWAY

     Else                          Return INDETERMINATE
```

End TEST_WARNINGS subprocess.

****

****

CREATE_PTL(TTF)

STRUCTURED_ENGLISH DESCRIPTION

****

```
Make TRACK_FILE point to AGED_IN_FILE:
     TRACK_FILE.TPTLP: = AGED_IN_FILE

Make AGED_IN_FILE point to TRACK_FILE:
     AGED_IN_FILE.PTTFP: = TRACK_FILE.

Set up the AGED_IN_FILE time and priority fields:
     AGED_IN_ FILE.PTOA: = TRACK_FILE.TTOA,
     AGED_IN_FILE.PPRIO: = TRACK_ FILE.TPRIO

Insert AGED_IN_FILE at the head of the PTL time_of_arrival  ring
     (procedure PRIOTHLIST.INSERT)

Call PRIOTHLIST.RE_PRIORITIZE to insert AGED_IN_FILE in its proper
     place in the PTL priority ring

End of CREATE_PTL(TTF) Subprocess
```

****

```
**************************************************

                CREATE_PTL (TCF)

          STRUCTURED_ENGLISH DESCRIPTION

**************************************************

Make CORRL_FILE point to AGED_IN_FILE:
     CORRL_FILE.CPTLP: = AGED_IN_FILE

Make AGED_IN_FILE pont to CORRL_FILE:
     AGED_IN_FILE.PTCFP: = CORRL_FILE

Set up the AGED_IN_FILE time and priority fields:
     AGED_IN_ FILE.PTOA: = CORRL_FILE.CTOA
     AGED_IN_FILE.PPRIO: = CORRL_FILE.CPRIO

Insert AGED_IN_FILE at the head of the PTL time_of_arrival ring
     (Procedure PRIOTHLIST.INSERT)

Call PRIOTHLIST.RE_PRIORTIZE to insert AGED_IN_FILE in its proper
     place in the PTL priority ring.

End of CREATE_PTL(TCF) subprocess

**************************************************
```

5.2.3.6. The DECIDE_REACTION Process. In a full-up Threat_ Resolution_ Module (TRM) functioning within a realtime version of the entire VIDS software suite, DECIDE_REACTION will be a table-driven process that will make recommendations to the crew on how to react to the external objects currently perceived to be threatening the VIDS vehicle. The stage has been set for this by the present DECIDE_REACTION process which receives the same input as the future process (the Prioritized_ Threat List -- PTL), but makes no recommendations.

DECISION_REACTION is called by THREAT_RESOLUTION when the TRACK_FILE currently being processed or the correlated object to which it belongs achieves aged_in status. Thus, each call on DECIDE_ REACTION provides it with a pointer to an AGED_IN_FILE. This pointer is placed in a record on a chained list accessed via THREAT_RESOLUTION output argument, OUTPUT_PTR. The REACTION_MANAGEMENT process of the TESTBED will list those members of this output chain which exhibits one of the non-trivial special warnings affixed to the AGED_IN_FILE by the TEST_WARNINGS subprocess of the AGE_IN process. Each AGED_IN_FILE thus exhibited will have its STATUS field changed to COMPLETED; this change of status will affect the FIND_ROOM subprocess of TRACK and the AGE_OUT module.

5.2.3.7. The AGE_OUT Module. The AGE_OUT Module (AOM) is an independent software module closely related to the TRM that is called from the same external software level that invokes the TRM itself.

The TRM and the AOM are file-oriented modules. The major function of the TRM, as we have seen in the preceding paragraphs of this section, is to create, update, and maintain a set of dynamic data files which collectively represent a perception of the external battlefield situation. (The term "file", as used here means a structured collection of data records stored in the memory of the DMS computer, i.e., an internal file). The three principal types of dynamic data files are the Threat-Track Files (TTFs) created and maintained by the TRACK process, the Threat_Correlation Files (TCFs) created and maintained by the CORRELATE process, and the Prioritized-Threat List (PTL) created and maintained by the AGE_IN process. The function of the AOM is to remove old or spent files from the dynamic data base. Thus, the relationship between the TRM and the AOM is that they share the data structure and the procedures/functions which handle the various data file types listed. The internal details of the dynamic data base and its handlers are provided in Section A of this document.

The AGE_OUT_MODULE presented in structured_English below is based on elimination of files that have not been seen for a period of time exceeding the SENSOR_INDEXed entry to the STATIC_DATABASE array, AGE_OUT_TIME. A single static data item, MIN_AGE_OUT_TIME, represents the minimum over all sensors of the entries in the AGE_OUT_TIME array. Age_out should also be based on taking cognizance of a file's reaction STATUS (NONE, WAITING, ACTIVE, or COMPLETED), but this refinement has been postponed to a future version.

```
**************************************************

                    AGE_OUT_MODULE

              STRUCTURED_ENGLISH DESCRIPTION

**************************************************

First of all, if the Prioritized-Threat List is empty, there is
simply nothing to do here, so return.  (Boolean value of
PRIOTHLIST.EMPTY is true).

Otherwise, we set up for a loop over the entire PTL in old to young
order, i.e., over the time_of_arrival ring:

     THIS_ITEM: = PRIOTHLIST.TAIL (PTL_GLOBAL_TOA)
     LAST_ITEM: = PRIOTHLIST.HEAD (PTL_GLOBAL_TOA)

Repeat the following loop until one of its exit conditions is met:

     Set AGE to the difference between the present time (CLOCK_
     TIME) and the latest_time_seen of THIS_ITEM (THIS_ITEM.PTOA)

     Exit when AGE < MIN_AGE_OUT_TIME -- since we are moving in old
          to young order, no other PTL item will be able to age-out
```

```
Capture the pointer to the next oldest item now, before a
     possible delete of THIS_ITEM destroys the pointer:
     NEXT_ ITEM: = THIS_ITEM. (backward pointer for the time
     ring)

Now, determine whether THIS_ITEM ages_in a correlated object or
     an uncorrelated TTF.  Except for testing the other exit
     condition and moving on to the NEXT_ITEM, the rest of
     this loop is spent inside the then and else clauses of
     this decision:

If THIS_ITEM ages_in a correlated object (THIS_ITEM.PTCFP not
null), then Let COR_FIL: = THIS_ITEM.PTCFP
             Set MAX_AGE: = zero_time (TZERO a constant from
                            TIME_PACK).

             Let CITEM be the COR_PTR to the first link on the
                 chain of COR_RECs that record the TTFs
                 belonging  to this correlated object:  CITEM:
                 = CORRL_FILE.COR_FIRST

             Repeat the following loop until its exit condition
             is met

                Set TRIAL_AGE to the AGE_OUT_TIME of the
                    sensor that pertains to the TTF pointed
                    to by the COR_ITEM field of the COR_REC
                    pointed to by CITEM:
                    TRIAL_AGE  := AGE_OUT_TIME( CITEM.COR_ITEM.
                    SIPDATA.SENSOR_ID)

                If TRIAL_AGE > MAX_AGE
                then Reset MAX_AGE to the value of TRIAL_AGE

                Exit when the pointer to the next link (CITEM.
                     NEXT is null

                Otherwise, set CITEM to CITEM.NEXT

             End loop

MAX_AGE is now the maximum AGE_OUT_TIME associated with the corre-
lated object (COR_FIL_).  If the AGE of THIS_ITEM is > = MAX_AGE,
we delete all associated TTFs, the TCF and the PTL in that order:

If AGE > = MAX_AGE

then Let CITEM be the COR_PTR to the first link of the chain of
          COR_RECs that record the TTFs belonging to this
          correlated object:  CITEM: = CORRL_FILE.COR_FIRST
```

```
        Repeat the following loop until its exit condition is met

            THREATFILE.DELETE the TTF pointed to by the COR_ITEM
                 field of the COR_REC pointed to by CITEM:
                 CITEM.COR_ITEM

            Exit when the pointer to the next link (CITEM.NEXT)
                 is null

            Otherwise, set CITEM to CITEM.NEXT

        End loop

        CORRELFILE.DELETE (COR_FIL  )
        PRIOTHLIST.DELETE (THIS_ITEM)

      Else -- THIS_ITEM ages_in an uncorrelated TTF

        Set MAX_AGE to the AGE_OUT_TIME of the sensor that
            pertains to the TTF that THIS_ITEM ages_in:
            MAX_AGE: = AGE_OUT_TIME (THIS_ITEM.PTTFP.SIPDATA.SENSOR_ID)

        If MAX_AGE > = AGE (of THIS_ITEM),
        then THREATLIST.DELETE (THIS_ITEM.PTTFP)
             PRIOTHLIST.DELETE (THIS_ITEM).

      End of the major If statement inside this loop

      Exit when THIS_ITEM = LAST_ITEM

      THIS_ITEM: = NEXT_ITEM

    End of Old_to_Young loop over the PTL

    End AGE_OUT Module

        ******************************************************
```

5.2.3.8. Miscellaneous Packages. The TRM defines or shares with the testbed a number of packages which have shown up in the foregoing paragraphs either peripherally or anonymously. Their contributions are nonetheless important and need to be understood. This paragraph describes those packages not described in Section 5.2.1 or Section 5.4.

SETS_PACK is a super_package that contains two packages, SENSOR_SETS and EMITTER_SETS. These two packages are 100% identical, differing only in the Ada type of the set element, and could have been implemented using Ada generics had been available in our compiler. Each package defines a set of sensors/emitters as a vector of Boolean flags, one for each possible element, that equal true if the element is present in the set, otherwise false.

Each package also defines the null_set, a full complement of set composition functions (union, difference, intersection, complementation), tests for empty set and set membership, and a function for creating a set out of a list of literal set elements. These capabilities are used to define the SENSET and EMISET fields in the record element of the CAN_BE_CORRELATED array in the STATIC_DATABASE and these fields are used extensively in the CORRELATE process.

5.2.3.9. Lessons Learned -- The Threat Resolution Module and Ada. Lessons learned in the application of Ada to the design of the dynamic and static data bases were given earlier in Sections 5.2.1.6 and 5.2.2.3, respectively. These concluding remarks emphasize these again by reference and add to them sundry observations:

In developing the TRM we experienced at every turn problems connected with missing features or incomplete features. At first, we simply devised workarounds, and went on our way. After a while, we learned to insert "KLUDGE" notices directly into the comments to mark places where we were forced to provide workarounds. Thus, we have laid down an indicative, if incomplete, record of these difficulties. Looking for and reading these KLUDGE notices is worthwhile.

The Ada UNCHECKED_CONVERSION feature was useful in several places. One such place is in the dynamic database packages. Consider THREATFILE which deals with TTF blocks: the only TTF block pointer fields which are not elements of the ring_pointer array, are pointers that point to TCF and PTL blocks. How then to provide a pointer from keeping TTF blocks on the chain of available blocks, without forcing an array indexing operation and without providing a pointer field specifically for that purpose? The answer: Use the TCF pointer as a TTF pointer via UNCHECKED_CONVERSION.

The invisible work that lies behind the development of the TRM and the myriad cycles of testing, correlation, and recompilation could have been simplified if we had been able to compile a package's body separately from its specification. A host of aids that one expects in a mature compiler -- conditional compilation, for example, would have been useful.

The real lessons lie ahead as we move toward a real time design with a fully validated compiler. We will face issues that the primitive state of our compiler allowed us to ignore: space compression using representation specifications, performance -- full checking or not, dynamic deallocation, the numeric representation of data, and concurrent operation of tasks.

We may someday look back on the lessons we have learned here as our elementary school years; our higher education lies ahead.

5.3. Code Generation Using Ada as the Programming Design Language.

For the development of the VIDS software, an Ada-based program design language (Ada-PDL) was used. This Ada-PDL consisted of Ada control structures, type declarations and compound statements with English comments to specify the processing requirements of each module. The control structures used were the declarations for procedures, packages, and functions. Tasks were not used because they were not fully developed in TeleSoft Ada. The compound statements consist of the if, case and loop statements.

5.3.1. What was done. The Ada-PDL was used at all levels of the design process. At the top level, the modules of the system and their external interfaces were specified. Each module, except the main program module, was defined as a package. The interfaces between modules were defined using type declarations, global constants, and subprogram specifications. No global variables were used. Comments were used to describe the purpose of all objects declared in the specification. In the interface, all external properties of the package are fully defined. The top level design could be compiled for interface verification.

During the detail design phase, we defined the implementation of the external subprograms. In the bodies for the subprograms, the steps to be done were outlined using compound statements and structured English comments. We avoided premature coding but did use subprogram calls to indicate transfers to other subroutines. This showed how the parts of the system fit together. Since the interfaces were fully defined, we determined exactly what information was required and what information was returned. If it became apparent that more information was required by a particular module, the appropriate changes were made to the intertree and the interface was recompiled. At this time, additional modules were defined consisting of routines which are used by some modules at a local level.

For the coding phase, we took the detail design and filled in the Ada code required to do the steps described in the comments. Further implementation decisions were made but we avoided all changes to the interfaces unless no other alternative was available.

5.3.2. What Ada Features Were Used and How.

5.3.2.1. Packages and Subprograms. Packages were used to form separately compiled program modules which were used as building blocks to piece the system together. We used packages for three applications.

The first was to form a collection of commonly used data types and constants. This enabled us to form a library of terms used throughout the system. One use of the constant declared in these packages was to define the range limits for type definitions. Modification to the limits became a simple matter of changing the constant. All references to the limits were by means of the constants so only one change was required for modifications. We did not use packages to declare global variables. This eliminates any questions about the accuracy of the values of global variables.

Another use of packages was to form groups of related programs. Only the main operating executive was a separately compiled procedure. The major processing modules for the testbed (the TRM, the CLOCK_MANAGER...) were defined as procedures in a library package. An implementation restriction of Ada required this design approach. Other library packages were defined for subprograms used by the testbed modules. Each process was broken down into smaller components until the simplest procedures were created. They were then grouped together to form library packages. The library packages were kept small with 100200 lines of code. This was done to reduce compile and debug times.

The third application of packages was to define abstract data types and the functions which act on these types. An example of this application is the package to define time. The unit of measure for time is defined by the type TIME. Functions for adding to time, multiplying time by a number and to tell time were all defined as part of the same package. However, due to an implementation restriction, we were not able to use the privacy feature of Ada packages. So, we were unable to protect time variables from misuse. We wanted to declare a deferred constant for time TZERO but were unable to because deferred constants were not used in TeleSoft Ada at the time of this research. This restriction did not impede the development, it just made the code less structured.

Packages were a valuable tool for data and information hiding. We were able to hide the implementations of the external procedure from the user. We were also able to hide the implementation of the data structure and thus control all access to the data by means of the external procedures declared for that structure. This provided us with a secure database without requiring extensive security checks for each access request.

5.3.2.2. Tasks. Tasks were not used in this development for several reasons. TeleSoft Ada did not fully implement all tasking features. It excludes entry familes and task types. These are two very important features of Ada. Another reason to reject tasking is the scheduling algorithm for the execution of tasks. A task would have full control of the CPU time until an event such as a rendevous occurs. This means that tasks do not really operate in parallel and one task could monopolize the computer. We must study the feature before we have to rely on it in the system. Since the emphasis of this effort was to test the algorithms employed in the TRM, and these algorithms could be transferred to a task implementation, we postponed the use of Ada tasking to a later development. We foresee a need for full tasking abilities in further development of the VIDS-FDM.

5.3.2.3. Input/Output. The interface of the VIDS testbed and the environment model was through static data files. The environment model is executed first to create a data file of Sensor Input Packs (SIPs) for each sensor. The package SIP_I/O in the testbed then reads these files and at the appropriate time will send the SIP to the TRM. SIP I/O is the only package in the testbed to access the SIP files. This interface requires

use of the Input/Output features of Ada. Because of the way by which we access the data files, we used the DIRECT_IO package exclusively. This enables us to read the files starting at any location in the file.

5.3.2.4. Other Features. One disadvantage of strong typing is illustrated by the lack of generics. Because each type is different, we had to write separate routines for objects of different, but similar, types. For example, a routine to add a component to a list would have to be written for each type of list. A general purpose generic routine could have been employed had this ability been provided in the compiler.

Variant records were restrictive because we had to know the exact variation for all objects declared. This eliminates the possibility of a local general purpose variable of the variant type in a subroutine. There are times when a routine should operate on objects and not be concerned with the exact variation. Generics may be the answer for this problem.

Generics are a valuable feature of Ada which can greatly simplify code. The lack of this feature reduced the convenience and value of other Ada features. A large portion of source code could have been eliminated if we had been able to declare program templates.

The package feature is a convenient way of breaking code into manageable portions. Use selected component notation when referencing components of a package. Avoid the use statement except for the most obvious cases at the lowest level of code. This requires more typing and longer references but it avoids confusion during the debugging in a large system. A programmer intimately familiar with a code module may be able to keep all the names straight but after a time, even this programmer will forget where something was defined. Also, selected component notation enables a person new to the code to more quickly understand the code. Avoiding the use statement is unpopular. In the long run, however, the advantages far outweigh the disadvantages.

Exceptions can be a source of confusion when the program bombs on an unhandled exception. It was best to put an exception handler in all subprograms and packages. The handler would identify the area where the exception was raised. We used the pragma SOURCE_INFO and the procedure SYSTEM.REPORT_ERROR to identify which exception was raised. This was valuable during the debugging stage. We also enclosed all input and output requests with a block containing an exception handler. All problems that surfaced during the I/O process were handled locally and we could restart the I/O request as needed.

Some of the predefined exceptions covered too wide a range of errors so that identifyihg a particular mistake was not always easy. The rules for propagation are complex and could confuse a beginning programmer. The propagation of an exception could also add some confusion to the process of determining what error was committed and where it occurred.

5.3.3. Difficulties Encountered. The difficulties encountered during development were frequently due to implementation restrictions of the partial compiler. However, these difficulties were identified early and the design was structured around the restrictions so that when it came time to code the design, elaborate workarounds were not required. The most common difficulty encountered is the lack of a feature which would have simplified the problem. Sometimes the lack of one feature would affect the use of another. Full data abstraction was not possible because deferred constants were not provided in the compiler. So, we could not hide the implementation of a type if we needed a constant for that type.

The area that required the most debugging time was with input/output. These programs required an interface with the underlying run time operating system. The system did not do the functions required adequately. If we added to a pre-existing data file so that it required more space than had been originally allocated, the operating system would return an error indicator for the lack of space, even though space exists on another section of memory.

The run time operating system should also swap out out unused portions of code from RAM. We encountered difficulties when our program was so large that it exceeded the space in RAM and overwrote portions of code. Since not all code is always required, a paging technique could have been employed. Another problem that we encountered was finding the cause of this error. We needed a debugger to access the code and determine how the code was loaded.

The underlying development operating system must support Ada. We ran into problems with the ROS system because the filer could not dynamically assign new space at run time to the data files created in an Ada program. A full set of development tools is required. Because the dependencies of modules can become complex, a tool is required to keep track of the dependencies. This tool must also make a distinction between the specification and body of a package because dependent modules only need to be recompiled if the specifications change.

5.3.4. Summarizing Ada as a PDL. Overall, Ada served our purpose very well. We could use Ada at the highest level without crippling our design effort. The transition through the design levels was smooth with minimal retracing of steps as the design became more complete. Ada promotes a black-box programming technique which enabled us to set aside modules whose requirements were not fully known. It also facilitates dividing the modules among several programmers. Each programmer can work independently of the other programmers. One person manages the interfaces between modules so that they grow and change, and the changes are done in an organized and controlled manner. This provides order to the otherwise cumbersome requirements of configuration management.

Debugging and system integration are also facilitated by the Ada design. The interface can be debugged separately from the implementation of the modules. The interfaces are debugged first so each programmer knows from the outset what information is available to his module and what information he must supply. The bodies of the modules are debugged using special test routines that provide a variety of possible data inputs. When the system is finally put together, there is a higher confidence level in the individual components of the system.

There were several advantages to using an Ada-based PDL. It enabled us to fully use the features of Ada such as strong typing and separate compilation. The Ada PDL provided a mechanism for describing and defining the system which could easily be incorporated into code. We were able to describe the abstract concepts for the system using the Ada terminology and later to code these abstract concepts directly. Our method of building the modules through each stage of development provided us with a precommented skeleton to be used for the coding of the software. This insures that the code is well documented and is consistent with the design documents. The Ada PDL fully supports a modular design approach so that modules which required more thought could be postponed until later. It also defined the interfaces early in the development and kept them relatively rigid so that the interface problems encountered during system integration were eliminated.

As with any language, bad programming practices create bad programs. Careful thought must be given to the definition of interfaces. This means that more time and effort is spent at the top-level design phase. But because so much care is taken, the next phases require less time. The applicability of the various Ada features must be studied carefully to choose the best implementation.

The Ada language is complex. A knowledge of a structured higher order language such as PASCAL is valuable to the student of Ada. Ada can be taught in a modular fashion starting with a basic subset of features which is built on as the student progresses. The full language should be taught and used. The training of Ada programmers is crucial because not only must the syntax be taught, but the use of the language must be taught. FORTRAN programming techniques are not appropriate in an Ada environment.

The result of well written Ada code by well trained Ada programmers is easily maintainable code. Modifications become more simple because the implementations of packages can be upgraded without affecting the overall system. The program can grow over time. New abilities can be added to the program easily by defining new modules. These new modules can use existing modules without requiring a recompilation of the entire program.

5.4. Testbed Design.

5.4.1. Introduction. The testbed is a collection of separately compiled modules. The TRM drops in to the testbed to form the test system. The full system is shown in Figure 5-2. This section describes the library packages and testbed software. The TRM and its support packages have been described in Section 5-1.

5.4.2. Math Library. In order to carry out the functions of the TRM, it is necessary to provide a library of mathematical functions. We adapted the algorithms developed by William J. Cody, Jr., and William Waite. (See References.[1]) The functions provided are briefly described in this section.

5.4.2.1. MATH_TYPES. MATH_TYPES provides the definition of a VECTOR of floating point numbers whose dimension is not specified. This definition is used extensively to define tables in the STATIC_DATABASE and to define constants for the polynomial approximations to the few transcendental functions used in the TRM (chiefly in the TRIG and STAT_FUNC packages). MATH_TYPES also provides a function, POLY_VAL, which computes the value of a polynomial given the argument X and the VECTOR of coefficients.

5.4.2.2. ELEM_FUNC. ELEM_FUNC provides SQRT (square root) used in ANALYZE_MOTION, INDEX_LOOKUP used by ASSESS_LETHALITY, and NEW_AVERAGE used by UPDATE_TTF. All these users are in package TRACK_AIDS.

5.4.2.3. TRIG_FUNC. TRIG_FUNC provies functions for the sine (SIN) and cosine (COS) with argument in degrees; two forms of arc-tangent with value returned in radians or degrees; and constants for degrees <--> radian conversion, and PI_OVER_TWO. SIN and COS are used in ANALYZE_MOTION and in the MOTION_HISTORY procedure of TEST_WARNINGS (AGE_IN).

5.4.2.4. STAT_FUNC. STAT_FUNC provides randomly distributed variables from the U [0,1] and the N [mu, sigma] probability density functions. These are not used directly in the TRM are useful in devising various ad_hoc drivers for the TRM or parts thereof during development. These functions were used by TESS to generate the SIP data.

5.4.3. Common Database Declarations. The functions of a number of packages is to provide definitions of data types that are used in many places. These packages are GEN_TYPES, SIP_PACK, REAC_TYPES, and TRM_TYPES. TRM_TYPES has been discussed extensively in Section 5.2.1. The remainder of these packages are described here.

5.4.3.1. GEN_TYPES. GEN_TYPES provides the basic definitions of the sensor and emitter related enumeration types including SENSOR_TYPES (with subtype SENSOR_INDEX) and EMITTER_TYPES (with subtype EMITTER_INDEX). It also defines METERS, DEGREES, and RADIANS as subtypes of the predefined floating point. The types of DEGREES and RADIANS should have been included in MATH_TYPES but the system design for GEN_TYPES had already been completed before we realized the need for a math library.

5.4.3.2. SIP_PACK. SIP_PACK provides the declarations for the input data. The two major declarations are for the record types SIP_ RECORD and INPUT_ DATA_BLOCK. The SIP_RECORD has been discussed in Section 5.2.3.2.

The INPUT_DATA_BLOCK defines the components of a linked list of the input data for the TRM. Each list component has two subcomponents: a SIP_RECORD and a pointer to the next input record. This linked list design enables us to have a variable length list of input records. In theory, we could have an infinite length list. However, in practice, this was not possible. A limit of 30 input records was set. This is due to the lack of garbage collection (reclaiming and collection of unused or deallocated space).

5.4.3.3. REAC_TYPES. REAC_TYPES provides the definition of the enumeration types REACTION_STATUS and WARNINGS and the THREAT_INFORMATION_BLOCK. REACTION_STATUS is used to provide a field in the PDL block that indicates the reaction status of an object. WARNINGS was exhibited extensively in Pages 91 through 94.

The THREAT_INFORMATION_BLOCK defines the components of a linked list to contain the output data from the TRM. The components of the output data consist of five elements:

SIP_DATA	:	SIP_RECORD
TRM_TOA	:	TIME
PRIORITY	:	FLOAT
MESSAGE	:	WARNINGS
NEXT_THREAT	:	(Pointer to next output record)

We imposed a limit of 30 elements of this linked list due to the lack of garbage collection.

5.4.4. TIME_LIBRARY. The TIME_LIBRARY consists of the single package TIME_PACK which defines the type TIME and the associated functions of TIME. This is a poor design because the number of code lines in the package makes it less maintainable. It would have been better to have broken it up into smaller packages using the same design approach as the math library. However, it was not considered necessary to break up TIME_PACK for this early development. The following paragraph will describe TIME_PACK.

5.4.4.1. TIME_PACK defines the type TIME used extensively throughout the TRM and testbed and defines all the ancillary operations such as: addition (+), subtraction (-), multiplicaton (*), and inequalities (<, <=, >=, >). It also provides for converting time values to and from floating point, and it provides facilities for keyboard input and display output of time values. Further, it provides the function to increment a time to the next whole second used by the OPERATIONAL_EXECUTIVE. This functions to return the whole seconds portion of the fractional seconds portion of a TIME.

5.4.5. Input Buffer Support Processes. The INPUT_BUS_SIMULATOR relies on a library of support processes. These are subroutines which have been isolated for separate development. They include POLL_PACK, SIP_INPUT_PACK, BUFFER_INFO, BUFFER_SUPPORT, and BUFFER_PACK, and are described in this section.

5.4.5.1. POLL_PACK. POLL_PACK defines the polling matrix for the sensors. Also defined are two functions which return the sensor to be polled and the sensor at the top of the matrix. POLL_PACK defines procedures to insert a sensor in the matrix, establish a default polling matrix, switch to the next sensor in the list and print the poll table to the screen. The polling matrix is used as a circular linked list. A Boolean function is defined to test if a new polling cycle has started.

5.4.5.2. SIP_INPUT_PACK. SIP_INPUT_PACK defines the routine to read SIPs from the data file on disk. It also maintains a Boolean array to indicate when a file for a sensor is empty. Isolating this function enables us to change the method of conveying SIPs to the testbed at some future date.

5.4.5.3. BUFFER_INFO. BUFFER_INFO defines the buffer type and the buffers for each sensor. The size of the buffer is determined by an array variable. Most buffers hold 100 elements. The read position in the buffer is stored in an array and routines are provided to increment the read position or reset it.

5.4.5.4. BUFFER_SUPPORT. BUFFER_SUPPORT provides the routines to clear the buffer and to fill it. The buffer is filled once and SIPs are extracted at each polling period. When the buffer is emptied, it is cleared and refilled with the next sequence of SIPs.

5.4.5.5. BUFFER_PACK. BUFFER_PACK supplies the external interface to the buffer. The routines in BUFFER_INFO and BUFFER_SUPPORT are only used by BUFFER_PACK. BUFFER_PACK contains a routine READ_BUFFER to read a single SIP from a sensor's buffer, that occurs (has a time stamp) before a certain time. This procedure is responsible for determining if a buffer is empty and filling it if it is. This READ_BUFFER routine is called by the INPUT_BUS_SIMULATOR.

5.4.6. Input Bus Simulator. The INPUT_BUS_SIMULATOR declares a procedure, READ_DATA, to read and fill the INPUT_DATA_BLOCK for transmission to the TRM. The last SIP in the list is always a null SIP. If the buffer is empty for the current sensor, a null SIP is returned. The sensor to be polled and the polling time are determined by the OPERATIONAL_EXECUTIVE and passed to the procedure READ_DATA. A pointer to and INPUT_DATA_BLOCK is returned.

5.4.7. Clock Time Manager. The CLOCK_TIME_MANAGER provides a simulated clock. There are two external functions availble: (1) a request for the current clock time; and (2) a request to increment the current clock time.

Execution of the CLOCK_TIME_MANAGER is initiated by the OPERATIONAL_EXECUTIVE. All requests for the current clock time will be handled through the executive. This module will use the time declaration and functions provided by the TIME_MODULE.

Only the clock manager can change the clock time. Time will be measured in a time unit which must match with the declaration of type time. The clock time will either be incremented by an even interval (1/100 of a time unit) or it will be incremented to the next full time unit. The external interface of the CLOCK_MANAGER contains a function to return the current clock time and two procedures to increment the current clock time. One of these procedures will increment the clock time by an even interval; the other will increment to the start of the next time unit. The even interval will always be 1/100 of a time unit. This time unit will be consistent with the units of measure in the Time-Module so that no conversions will be required.

5.4.8. Reaction Management. The REACTION_MANAGEMENT package defines a routine to display on the screen the current clock time and the THREAT_INFORMATION_BLOCK. It is called after all the data from one sensor has been processed for the current polling period. The current block time and the THREAT_INFORMATION_BLOCK are supplied as input parameters by the OPERATIONAL_EXECUTIVE. The routine could be easily modified to print the output on the printer instead of the CRT screen but at this time the printer is unavailable due to technical difficulties.

5.4.9. Debug Library. The DEBUG_LIBRARY provides a number of utility programs that are useful in the debugging of the code. Some of these routines were so useful that the library remains a part of the testbed. The packages in the Debug Library are ENUM_IO, DEBUG_AIDS, PRINT_PACK, and DUMP_PACK.

5.4.9.1. ENUM_IO. ENUM_IO is a collection of procedures to print enumeration literals for the enumeration types SENSOR_TYPES and EMITTER_TYPES.

5.4.9.2. DEBUG_AIDS. This package contains routines to print a message, to pause execution and wait for user input before continuing, to convert a character to a digit and to clear the screen.

5.4.9.3. PRINT_PACK. PRINT_PACK defines routines to format and print the individual components of the threat track data base.

5.4.9.4. DUMP_PACK. DUMP_PACK provides procedures for dumping the entire contents of the TRM's dynamic database files: DUMP_TFF, DUMP_TCF, and DUMP_PTL. Each procedure accepts a single input argument specifying the particular ring to be dumped. For example, calling DUMP_TTF (TTF_GLOBAL_PRIORITY) dumps (nicely formatted) all the TTF blocks in descending priority order. It is used by the OPERATIONAL_EXECUTIVE to dump the files at the end of each polling period.

5.4.10. Initialization Processes. These processes are called at the start of the run to initialize the databases and system parameters. The packages for these routines are SET_UP and STDB_MAINTENANCE.

5.4.10.1. SET_UP. SET_UP contains a routine to determine the length of the test time interactively. It also defines an interactive routine to ask if the user wishes to modify the static database in the TRM, and if so, call the routines in STDB_MAINTENANCE.

5.4.10.2. STDB_MAINTENANCE. This package contains the routines necessary to interactively modify the static database used by the TRM.

5.4.11. Operational Executive. The OPERATIONAL_EXECUTIVE is responsible for controlling the system. It initiates all initialization code and calls each of the testbed modules in turn. It is responsible for insuring the proper execution of the TRM test run.

At the start of each run, the OPERATIONAL_EXECUTIVE must establish the polling matrix, determine the length of time for the test run, and initialize the TRM database. A procedure for modifying the database is also called to allow the user to interactively modify the data.

The OPERATIONAL_EXECUTIVE will read SIPs, send the SIPs to the TRM and print the results until either all the SIPs have been read or the test time has been exceeded. The subprograms for the INPUT_BUS_SIMULATOR, the TRM, and REACTION_MANAGEMENT are called in that order for each sensor for a given polling period. Then the threat track database is dumped and the time incremented to the start of the next poll period. This process is repeated until the end of the test period or until there are no more SIPs.

5.5. Integration and Test Results

5.5.1. Introduction. This section discusses the test run results and the integration of the testbed with the TRM. The Threat Environment Scenario Simulator (TESS) was used to produce a test environment. This section will first discuss the test environment, then the integration, and finally the test run results.

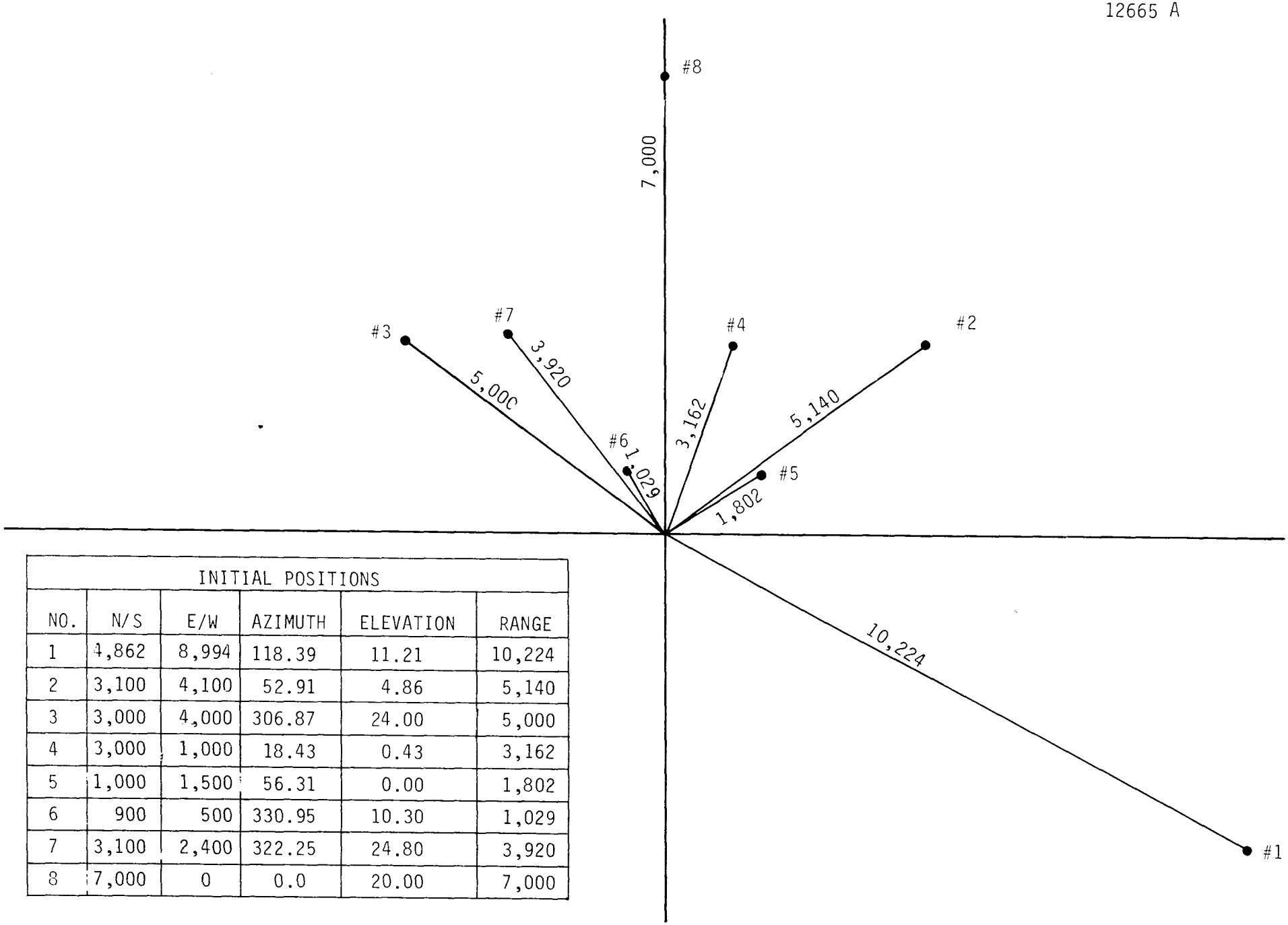
5.5.2. Test Environment. TESS produced SIPs for a 3.00 second scenario with eight active emitters. The environment is depicted graphically in Figure 5-16. The threats are identified by number and their initial positions are indicated on the table insert. Additional details about the threats are given in Table 5-2.

The scenario includes two moving threats; number 2 and 3. Threat number 2 represents a loiter pattern over a 200-meter square. The SIPs are generated for each corner and the center of the square in an alternating pattern. Threat number 3 is moving toward the tank at a rate of 68 meters per second. All other threats are stationary. Threat number 8 is reported on both sides of the North axis in order to test the logic of azimuth wrap-around. Separate SIPs are generated for each sensor. All SIPs include random measurement errors.

5.5.3. Testbed and TRM Integration.

5.5.3.1. Testbed/TRM Interface. The TRM consists of two external procedures which are called by the OPERATIONAL_EXECUTIVE in the testbed. The OPERATIONAL_EXECUTIVE supplies SIPs to the TRM and receives the final result. The SIPs are sent in clusters at regular polling periods. All SIPs active during the polling period for one sensor are sent in one group to the TRM. After the TRM has processed these SIPs, new ones are collected for the next sensor in the polling matrix.

12665 A



INITIAL POSITIONS					
NO.	N/S	E/W	AZIMUTH	ELEVATION	RANGE
1	4,862	8,994	118.39	11.21	10,224
2	3,100	4,100	52.91	4.86	5,140
3	3,000	4,000	306.87	24.00	5,000
4	3,000	1,000	18.43	0.43	3,162
5	1,000	1,500	56.31	0.00	1,802
6	900	500	330.95	10.30	1,029
7	3,100	2,400	322.25	24.80	3,920
8	7,000	0	0.0	20.00	7,000

Figure 5-16. Test Threat Scenario

TABLE 5-2. TEST ENVIRONMENT THREAT INFORMATION

POINT	EMITTERS	START TIME	STOP TIME	MOTION
1	Laser Beamrider; Attack Helicopter; Millimeter Wave	0.85	1.32	Stationary
2	Laser Designator; Scout Helicopter; Millimeter Wave	0.75	2.50	Loiter 200 meter square
3	Laser Rangefinder; Scout Helicopter	0.10	1.51	Diagonal 68 meters/ second per quarter-second
4	Laser Rangefinder	0.00	0.50	Stationary
5	Optical	0.50	1.00	Stationary
6	Optical	1.50	2.75	Stationary
7	Millimeter Wave	1.35	2.10	Stationary
8	Scout Helicopter; Millimeter Wave	2.25	2.75	Stationary

The TRM processing results are reported after each call to the TRM by the REACTION_MANAGEMENT module. After all the sensors have been polled, the threat tracking files are dumped. These files consist of the THREAT_TRACK_FILE, THREAT_CORRELATION_FILE, and PRIORITIZED_THREAT_LIST.

5.5.3.2. Integration. The integration process went smoothly. Because of the modularity of the system, we were able to integrate the separate components of the testbed and the TRM independently of the TRM testbed integration. So, the only integration required was between the external routines of the TRM and the testbed. This paragraph addresses the problems encountered during integration. It also addresses the advantages of using Ada as reflected during the integration effort.

a. Problems. The TRM and the testbed were developed separately by separate programmers. A specially devised driver was used to test the TRM before integration. In the testbed, a dummy module replaced the real TRM. When it was time to integrate, it was a simple matter to replace this dummy with the real TRM code. It does not require recompilation to do this substitution. However, due to the compiler deficiency, it was necessary for us to regroup the source into larger packages (fewer modules) and recompile the entire system. This effort took approximately five labor days. It was by far the most serious integration problem that we encountered.

Other problems arose due to assumptions made in the TRM about the input data that were not accurate. One such assumption concerns the TIME_SENSED field of the SIP. The value assigned to this field is the time that the sensor reported the SIP to the testbed and the SIP was placed in the buffer. The TRM expected the value to be the time that the input buffer was read and the SIPs passed to the TRM. Initially, this time interpretation disparity caused the TRM to end abnormally. The sites requiring correction were recognized and appropriate steps were taken. As the integration proceeded, sites where gross errors in the results were produced were recognized and corrected. In the final version, a subtle and hard-to-detect error remains; it is noted in the Results section.

b. Advantages of Ada During Integration. The modularity of Ada code enabled us to substitute revised code without recompiling the whole system. The linking of code modules occurs at run time so the most recent version was always being used. We did encounter a problem executing code after a large number of substitutions had taken place. To correct the problem we had to recompile the entire testbed/TRM system. A complete system regeneration should be performed on a regular basis to avoid any such problems.

A further advantage of Ada modularity during integration was that it generally enabled the integrators to localize errors to the package level rapidly. This ability was enhanced by judicious use of Ada's superior exception handling capability.

5.5.4. Test Run Results.

5.5.4.1. Introduction. The results reported here consist of comments on a test run printout submitted with this report and whose pages are headed as follows:

Oct 13 16:07 1984 Oct 23 testbed output page (pp)

where the symbol (pp) denotes a page number in the range of 1...35.

5.5.4.2. Overview of Results. This subsection briefly notes the abilities of the TRM demonstrated by the printout -- what we have accomplished. It also comments, in passing, on several TRM capabilities not shown and on the time sorting error noted in handwriting on the printout.

The referenced printout shows the operation of the TRM upon 56 SIPs (Sensor Input Packets) fed to it by the testbed over a simulated time period of four seconds. The input is divided into three major bursts of 16, 25, and 15 SIPs with a priority-ordered dump of the dynamic data files occuring after each burst. (A dump of these files also occurs before the first burst, but shows, as expected, that each file is empty.) The dynamic data files consist of the Threat Tracking Files (TTF), the Threat Correlation Files (TCF) and the Prioritized Threat List (PTL -- also termed "Aged_In Files"). It is chiefly upon these file dumps that we rely in stating the following conclusions about the operation of the TRM.

a. The TRM is able to correctly discern a "new guy". This is shown by the hand-annotated, one-to-one matchup between the input SIPs which are followed by in-process messages that read

"MATCH: Return with FLAG = Genuine_NG"

and the TTFs.

b. The TRM is able to correctly discern that an input SIP represents something that has been seen before. Each such SIP is followed by an in-process message that reads

"MATCH: REturn with FLAG = MATCH_WOC"

where WOC denotes "without_change." An audit of the "AGE_IN_COUNT" file of each TTF shows that each input having been seen the correct number of times -- once as a "Genuine_NG" and subsequently as a "Match_WOC."

c. The TRM is able to correctly discern that an input SIP bears a close, but not-quite-close-enough resemblance to an existing TTF. This occurs once at input SIP number 28 which is followed by the in-process message

"MATCH: Return with FLAG = Possible_NG"

This state triggers software which attempts to answer the question: Is the discrepancy due to motion? In this particular run, the answer returned is "no," i.e., the apprent discrepancy is not a discrepancy at all, and the track file and input SIP represent different objects -- the SIP is a "new guy." Other runs have demonstrated the TRM's capability to reach the opposite conclusion.

The TRM can also discern a fourth state in between the last two ("Match_WthC" -- with change), but this is not demonstrated on this particular run.

d. The TRM can do cross-sensor correlation, i.e., recognize that inputs from different sensors come from the same location. This is noted by two kinds of in-process messages:

"CORRELATE: New correlation file created"

"CORRELATE: Track File added to existing correlation"

As noted elsewhere in this report, this correlation is based on azimuth matching only and would have failed if the apparent correlated tracks had differed in other measurable parameters.

e. The TRM is able to "age_in" both individual TTFs and TCFs according to three rules:

- o Lethality exceeds a minimum;
- o Age_In_Count exceeds a minimum;
- o A special warning flag is raised. (Applies only to the correlation of TCFs).

The TRM can also handle the case wherein correlation occurs after the individual TTFs which are being correlated have aged_in. This is not demonstrated in this particular run.

f. The TRM can correctly update all fields in the various files and to maintain all files in their correct orders on their priority and azimuth rings. The PTL is not correctly ordered on its time_of_arrival ring. This is because the TRM was written under the expectation that SIPs would be input with their TIME_ SENSED fields in non-decreasing order (as if they were time_stamped upon entry into the TRM), but the testbed supplies time values for these fields which represent the times seen by the sensor supplying the data, and the polling regime followed by the testbed allows SIPs to violate time monotonicity. This can be corrected by supplying the PRIOTHLIST package with a procedure to correctly insert PTLs in time order on the time_of_arrival ring, and calling this procedure from each of the two overlays of CREATE_PTL (in AGE_IN_PAK) in place of the currently coded statements:

```
INSERT (AGED_IN_FILE, T_OF_ARR, BEFORE, P_HEAD);
```

The only consequence of this error is that the Age_Out_Module will not work correctly, but this module is not exercised by the current version of the testbed.

5.5.4.3. Examination of Test Run Printout

a. A Quick Walk-Through. This subsection provides a brief and coarse guide to the referenced printout:

Page 1-2: The 33 messages which begin "Elaborate.." are printed out in the elaboration section of each of the combined TRM/Testbed packages (some are shared).

Page 2: The testbed's Operational_Executive asks the user for some information and the user responds.

Page 2: A dump of the three major dynamic files shows that they are all empty.

Page 2-5: SIPs 1...16 are input.

Page 5-10: First dump to files:

pp 5-9: TTFs
pp 9-10: TCFs
pp 10: PTL (single entry)

Page 11-15: SIPs 17...41 are input, on p. 14, a brief excursion is taken through REAC_MAN between SIPs 33 and 34.

Page 15-23: Second dump of files:

pp 15-20: TTFs
pp 20-22: TCFs
pp 22-23: PTLs

Page 23-26: SIPs 42-56 are put.

Page 26-35: Third and last dump of files:

pp 26-32: TTFs
pp 32-33: TCFs
pp 33-35: PTLs

b. Detailed Commentary on Results. This subsection provides detailed commentary on places of interest in the printout.

Page 2: We note, with relief, that the very first SIP input is declared to be a genuine_NG (New Guy). As a matter of fact, five of the first seven are also.

Page 3: At SIP No. 7, we obtain our first correlation. Later inspection of the first TTF and TCF dumps shows that this correlation is between the object representing SIPs (1 ,3, 4) and that representing SIPs (7, 8, 9, 12).

Page 4: Two more fresh correlations are obtained; the cognizant SIPs are hand-annotated on the printout.

Page 5: Two correlations are obtained which refer to already existing correlated objects.

Page 5: Dump of the TTFs. The number of TTFs equals the number of "Genuine_NGs" (10) found in the SIP input prior to the dump, and that the summation of the AGE_IN_ COUNT fields over these ten TTFs equals the number of SIPs input thus far (16). The TTFs (as well as the TCFs and PTLs) are in descending order on priority.

Page 12: A "Possible_NG" is declard by the Match Subprocess at SIP No. 28. The signficance of this declaration and the ensuing motion analysis are explained in the annotation provided with the printout.

Page 15: The second dump of the files reflects the three additional "new guys" discovered in the second burst of SIPs, and the summation of the AGE_IN_COUNT fields is now 41. One of the three TCFs is now aged_in and this is because the correlated object is found to be illuminating the VIDS vehicle; two individual TTFs have also aged_in on the lethality rule.

Page 24: A new "Genuine_NG" is discovered at SIP No. 46. The two following SIPs match No. 46 and demonstrate that the match logic correctly handles the wraparound of azimuth from 360 to 0 degrees.

Page 25: SIPs No. 54...56 reconfirm the remarks on Page 24; this object also correctly correlates with (46, 47, 48).

Page 26: The third and final dump of the files is confirmed for correctness by the same techniques used for the first two dumps (pp. 5, 15).

Page 34: Analysis of the PTL's time_of_arrival (T_OF_ARR) ring pointers shows that time monotonicity is violated; inspection of the second dump shows that it happened there as well. The cause and cure of this problem are discussed above.

REFERENCES

1. SOFTWARE MANUAL for the ELEMENTARY FUNCTIONS, William J. Cody, Jr., and William Waite - Prentice-Hall, Inc., 1980.

2. MIL-STD-1815 - Ada Language Reference Manual.

GLOSSARY

DECLARATION - A declaration is the definition of an entity. The declaration will give the characteristics of the entity.

ENTITY - An entity is anything that can be named or denoted in a program. Objects, types, values, program units, are all entities.

OBJECT - An object is a variable or a named constant. An object can denote any kind of data element, whether a scalar value, a composite value, or a value in an access type.

PACKAGE - A package is a program unit that is used to describe groups of logically related types, objects, and subprograms whose inner workings are concealed and protected from the user. It can consist of two parts: a visible part and a private part.

Visible Part - The visible part of a package contains the entities that may be used from outside the package.

Private Part - The private part of a package contains structural details that are irrevelent to the user of the package, but that completes the specification of the visible entities.

PARAMETER - A parameter is one of the named entities associated with a subprogram, entry, or generic program unit. A formal parameter is an identifier used to denote the named entity in the unit body. An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation. A parameter mode specifies whether the parameter is used for input, output, or input-output of data. A positional parameter is an actual parameter passed in positional order. A named parameter is an actual parameter pased bynaming the corresponding formal parameter.

SPECIFICATION - A specification defines the identifier of a pro-gram unit and the parameters and interface requirements for other porgram units.

SUBPROGRAM - A subprogram is an executable program unit that is invoked by a subprogram call. There are two forms of subprograms: procedures and functions.

TYPE - A type characterizes a set of values and a set of operations applicable to those values. The values are denoted by either literals or by aggregates of the type and can be obtained as a result of operations.

DALMO VICTOR TEXTRON

Dalmo Victor Operations
Dalmo Victor Division of Textron Inc.

1515 Industrial Way
Belmont, Ca 94002-4095
Tel: (415) 595-1414
TWX: 910-376-4400

April 24, 1985
DV Letter No. 85-S8571

Commander
U.S. Army Tank-Automotive Command
Warren, MI 48090

Attention: DRSTA-ZSC

Subject: Contract DAAE07-83-C-R056
CLIN 0002 ELIN A002

Gentlemen:

In accordance with above subject, Dalmo Victor resubmitts twenty five (25) copies of the Final Scientific and Technical Report, Dalmo Victor reference number R-3811-10968A.

If you have any questions feel free to contact me at (415) 595-1414 extension 2622.

Sincerely,

DALMO VICTOR OPERATIONS
Division of Textron Inc

Ann Sabatini
Contract Administrator

cc: DRSTA-TSL
Warren, MI (1)

DCASMA-SF
San Francisco, CA (1)

DISTRIBUTION LIST

	No. of Copies
Commander US Army TK-Autmv Command ATTN: Countermeasures Func (AMSTA-ZSC) Technical Library (AMSTA-TSL) Warren, MI 48397-5000	 15 2
Commander Defense Technical Information Center ATTN: DDAC Bldg 5, Cameron Station Alexandria, VA 22314	12
Manager Defense Logistics Studies Information Exchange ATTN: DRXMC-D Fort Lee, Va. 23801	2